

版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！



Web开发经典丛书

Pro React

# React开发实战

使用React以组合方式构建复杂的前端应用程序

[美] Cássio de Sousa Antonio 著  
杜伟 柴晓伟 涂曙光 译

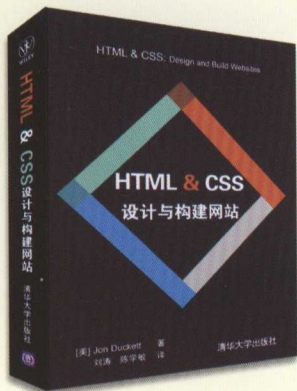


Apress®

清华大学出版社

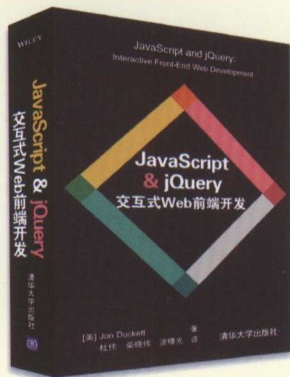


## Web开发经典丛书



不管你设计和建立新网站，还是想更好地控制当前网站，都可以在本书的指导下创建出用户友好、令用户赏心悦目的Web内容。我们知道，编码是一件令人望而生畏的工作，而本书却采用有别于许多传统编程书籍的新颖编排方式，将使你收到事半功倍的学习效果。

每一页都在短小精悍的示例代码的引导下，简明直观、直截了当地阐述一个新主题。本书还提供关于如何组织和设计网页的实用信息，以便帮助你创建充满魅力、易于使用的网站。学习本书不要求你具有任何经验！

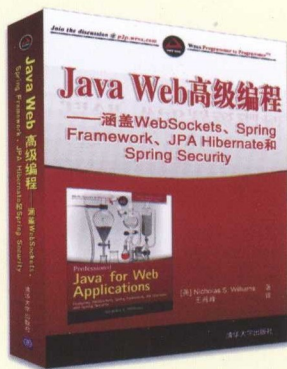


你是一名JavaScript新手？或是你曾经向自己的Web页面上添加过一些脚本，但想以一种更好的方式来实现它们？那么本书非常适合你。本书不仅向你展示如何阅读和编写JavaScript代码，还会以一种简单且视觉化的方式，教你有关计算机编程的基础知识。阅读本书之前，你只需要对HTML和CSS有一些了解即可。

通过将编程理论与用来演示JavaScript和jQuery如何被应用于流行站点之上的示例相结合，本书将教会你如何让网站更具交互性、吸引力、可用性。很快，你就能像一名程序员那样去思考和编写代码了。



JavaScript把Web从被动媒介转变为丰富、动态的交互式媒介。通过这本内容全面的入门图书，可学会JavaScript目前最常用的使用方式，学习利用最新的工具和技术来创建动态Web应用。本书讲解了如何高效地使用JavaScript框架、函数和现代浏览器，如何使用HTML5实现最有效的编码实践。本书主要内容包括：以最新的编码风格使用JavaScript、新的HTML5元素和相关API、使用JavaScript向Web服务器发出HTTP请求、分析常见错误以及调试和错误处理方法等。



Java成为世界上编程语言之一是有其优势的。熟悉Java SE的程序员可以轻松地进入到Java EE开发中，构建出安全、可靠和具有扩展性的企业级应用程序。编写本书的目的正是如此。

本书面向已经了解Java SE、SQL和HTML基础知识，准备将他们的Java编码技能提升到更高水平的程序员。软件开发人员可以按顺序阅读本书或者在遇到特定的编程问题时将某个章节用作参考。

Web 开发经典丛书

# React 开发实战

[美] Cássio de Sousa Antonio 著

杜伟 柴晓伟 涂曙光 译

清华大学出版社

北 京

Pro React

By Cássio de Sousa Antonio

EISBN: 978-1-4842-1261-5

Original English language edition published by Apress Media. Copyright © 2015 by Apress Media.  
Simplified Chinese-Language edition copyright © 2017 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2016-8577

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

React开发实战/(美)卡西奥·德·宗萨·安东尼奥(Cássio de Sousa Antonio) 著；杜伟，柴晓伟，涂曙光 译。—北京：清华大学出版社，2017

(Web 开发经典丛书)

书名原文：Pro React

ISBN 978-7-302-46197-5

I. ①R… II. ①卡… ②杜… ③柴… ④涂… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2017)第 019984 号

责任编辑：王 军 韩宏志

装帧设计：孔祥峰

责任校对：曹 阳

责任印制：宋 林

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质 量 反 馈：010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者：清华大学印刷厂

装 订 者：三河市漂源装订厂

经 销：全国新华书店

开 本：185mm×260mm 印 张：19.5 字 数：451 千字

版 次：2017 年 3 月第 1 版 印 次：2017 年 3 月第 1 次印刷

印 数：1~4000

定 价：58.00 元

产品编号：071425-01



# 译者序

本书的三位译者都有基于Microsoft SharePoint Server平台进行Web应用程序开发的经验。在SharePoint平台(特别是从SharePoint 2007 这个版本开始)上进行Web应用程序开发,主要使用的还是服务器端的开发技术,主要编程语言和框架是C#和ASP.NET。但从那时起,我们就已经不断在自己的Web应用程序中使用越来越多的JavaScript代码,并尝试着开始将(当时还算是比较新潮的)jQuery和其他前端库引入到自己的应用程序中。我们当时就预料到,JavaScript在未来必然将占据越来越重要的地位,对于Web应用程序开发而言,越来越多的服务器端技能都会被前端技能所替代。

但JavaScript的发展之迅猛,仍然超出了我们的意料。时至今日,不但JavaScript已经成为前端工程师的一件利器,由于Node.js的出现,JavaScript还逐渐渗透到服务器端(甚至桌面端)之上。前端工程师们开始尝试基于JavaScript来构建越来越复杂的Web应用程序,这给JavaScript提出了越来越大的挑战,随之而来的,就是异常活跃的JavaScript社区和不断涌现的JavaScript前端框架。这些框架的目标,都是为了以一种更简明直观的方式,将Web应用程序的复杂性进行隔离,然后以一种模块化方式,单独开发和推进整个应用程序中相对独立的部分。

React是众多JavaScript前端框架中的后起之秀。它诞生于Facebook,被Facebook开源之后,便迅速得到众多前端工程师的青睐。React社区也随之蓬勃发展,各种React的插件库不断地由社区成员开发出来,其中的佼佼者(比如本书中所介绍的React Router)也得到了大量关注并被广泛使用。

作者以一种循序渐进、深入浅出的方式,讲述 React 的方方面面。读者在阅读和学习的过程中,应该能很轻松地跟随着作者的节奏,逐个了解 React 的特性,并通过一个贯穿全书的示例应用程序,了解如何规划和构建出一个越来越复杂的 Web 应用程序。书中的示范代码也秉承了简洁易懂、突出重点的风格,能帮助读者迅速了解一个 React 应用程序的代码风格应该是什么样子。

巧合的是,在翻译本书的过程中,Microsoft 在 SharePoint 平台上发布了一个新的开发框架:SharePoint Framework。SharePoint Framework 是一个完全基于前端技术的 SharePoint 平台开发框架,其中所使用的默认 UI 前端库,正是 React。

本书全部章节由杜伟、柴晓伟、涂曙光翻译,参与翻译的还有梁祝权、钟凤华、毛士之、张杉杉、张文旭,在此一并表示感谢!最后,让我们向读者稍微介绍一下自己。杜伟(微博账号:@Erucy),软件工程师与日本动漫迷,现在一家互联网创业公司担任技

术总监，日常使用 JavaScript 和 Cordova 开发跨平台的 Web 和 Mobile 应用。柴晓伟(微博账号: @WindieChai)，软件工程师，现在一家大型招聘网站担任技术执行总监，日常使用 JavaScript、Node 和 C#开发各种 Web 前后端以及 Mobile 应用。涂曙光(微博账号: @kaneboy)，软件工程师，现在一家金融行业公司担任架构师，日常使用 C#/C++开发证券交易系统，同时还在使用 JavaScript 开发一些 Web 应用程序。我们都是微软最有价值专家，并已合作翻译过多本技术书籍。

Happy Reacting!

译者  
作于北京



## 作者简介



20 年前，Cássio de Sousa Antonio 使用一台 Sinclair Spectrum 开启了他的编程生涯，随后在巴西和美国成长为一名软件工程师和技术经理。他参与开发过微软、可口可乐、联合利华和汇丰银行等大公司的项目。他的初创公司于 2014 年末被收购。Cássio 目前担任技术顾问。你可在 Twitter 上关注他(@cassiozen)。

# 技术审阅者简介

Jack Franklin是一位讲师、作者和技术作家，他大多数的时间都在编写或讨论JavaScript。他在Pusher公司担任开发者布道师，他还是一名积极的开源贡献者。他是React的狂热粉丝，并在[www.javascriptplayground.com](http://www.javascriptplayground.com)上撰写了大量关于JavaScript的文章。你可在Twitter上找到他：@Jack\_Franklin。

Tyler Merry 是 Universal Mind 公司的 UX 技术专家，他专注于缩短创意和实现之间的差距。Tyler 通过试验来筛选处理所有问题。他坚信最快且最精准的解决方案就是积极地进行各种试验和非正式的测试。

Tyler 曾就职于可口可乐、索尼、辉瑞、宝洁、福特和 Vail Resorts，在工作中，他认识到了精确度和沟通的价值。他就职于一些早期的初创公司，去帮助它们增加迭代、速度和效率所带来的价值。

除了追赶最新的 Web 和 UX 趋势，Tyler 还把时间花在“少于四个轮子的车辆”上(自行车、摩托车和独轮车)，他也会去学习任何能够吸引他的技能，如编织、摄影或杂耍。





# 致 谢

我要感谢我的父母：Sergio 和 Dete，是你们给了我自由、独立和爱。  
还要特别感谢 Apress 的编辑人员，感谢你们在这个项目中给予我的信任，以及给予我的所有指导和耐心。

## 本书概要

第 1 章介绍了本书的动机和本书的架构。本章还介绍了 React 的架构，并让你能全面了解在 React 中开发应用程序的通用方法。

第 2 章首先介绍了 HTML5 的语义化文档、JavaScript 的 ES5 和 ES6 以及 CSS3 的进阶特性。第 3 章介绍了如何在 React 中设置一个应用，并展示了如何配置开发环境。

第 4 章讲解了如何通过组件的方式来构建一个应用程序的视图。本章还介绍了如何与后端的数据进行交互，并展示了如何使用 RESTful API 来构建一个应用程序。

第 5 章讲解了如何与用户创建表单应用。本章还展示了如何使用表单库（如 Formik 或 Yup）来验证表单数据，并展示了如何使用表单库来构建一个表单应用。

第 6 章讲解了路由功能，会教你如何使用 React Router 来构建一个单页应用（SPA），并展示了如何使用 React Router 来构建一个单页应用。

第 7 章讲解了如何与第三方 API 进行交互。本章还展示了如何使用第三方 API 来构建一个应用程序，并展示了如何使用第三方 API 来构建一个应用程序。

第 8 章讲解了如何与第三方 API 进行交互。本章还展示了如何使用第三方 API 来构建一个应用程序，并展示了如何使用第三方 API 来构建一个应用程序。

第 9 章讲解了 React 的测试方法，包括如何使用 Jest 来测试组件，以及如何使用 Enzyme 来测试组件的交互。本章还展示了如何使用 Jest 来测试组件，以及如何使用 Enzyme 来测试组件的交互。

# 前言

React 是一个用来创建组合式 Web 应用程序的开源库，由 Facebook 维护。自从公开发布后，这个库迅速风靡全球，并且围绕着它产生了一个生机勃勃的社区。

本书将涵盖 React 库的各个细节，并将讨论基于组合式模型来创建 Web 组件接口的最佳实践。React 库本身并不大，所以本书同时涵盖了 React 生态系统中的一些工具和库(例如 React Router 和 Flux 架构)，以便为读者提供创建完整应用程序所需的足够知识。

书中对每个主题的讲解都简洁明了，你将逐一了解到你需要掌握的各个细节，从而学会真正有效地使用它们。本书对 React 中最重要的那些特性的讲解，言简意赅、由浅入深，每个章节中还详细说明实际开发中可能面临的常见问题，并告诉你如何避免它们。

## 本书概要

第 1 章提供了大量的信息来让你起步，介绍基本的 React 配置，并让你能全面了解在 React 中如何组织用户界面。

第 2 章将深入介绍 JSX(这是 React 用来在 JavaScript 中声明组件标记的 JavaScript 语法扩展)，同时展示如何使用 React 的事件系统，以及 React 的虚拟 DOM 实现。

第 3 章讲述了如何通过使用组件的方式来创建一个完整的应用程序。你将学习 React 应用程序的数据如何在组件间流动，并深入了解组件(包括嵌套组件、公开一个 API、props、state 等知识)。

第 4 章讲述了如何为用户创建丰富的用户体验。你将学习如何在 React 的插件 CSSTransitionGroup 的帮助下实现动画效果，以及使用一个名为 React DnD 的外部库来实现拖放功能。

第 5 章讲述了路由功能。你将学习如何使用 React 社区中被广泛使用的 React Router，来管理 URI 和设置应用程序的访问点。

第 6 章向读者展现了 Flux 架构。你将学习这个架构的细节，它解决了哪些问题，以及如何将它集成到一个 React 应用程序中。

第 7 章讲述了性能调优。在该章，你将学习如何度量应用程序的性能指标。然后你将了解如何优化代码，使应用程序有更好的性能表现。

第 8 章讲述了 React 同构应用程序(或者称 React 通用应用程序，也就是在服务器上渲染 React)。这种技术可以实现更好的性能、搜索引擎优化和优雅降级(如果浏览器禁用

了本地 JavaScript，应用程序也能正常工作)。

最后，第 9 章讲述了测试。你将学习如何使用 React 的 Test Utils 来测试组件。还将学习 Jest，这是一个由 Facebook 创建的、最适合用来测试 React 项目的测试框架。

## 本书读者对象

本书主要面向使用诸如 jQuery/Backbone/Angular 创建前端应用程序且拥有一些经验的中级水平 JavaScript 开发人员，他们需要了解更好的工具和更多知识，来解决构建复杂前端应用程序过程中所遇到的越来越多的常见问题。

## 源代码

本书中所包含的示例项目的源代码，位于 Apress 网站上的 Source Code 区域。请访问 [www.apress.com](http://www.apress.com)，单击 Source Code，然后查找本书的书名(请使用英文书名 *Pro React* 来进行搜索)，就可以找到它们。另外，本书所有的示例代码和一些额外的实用性代码，都可在本书的 GitHub 页面上找到，网址是 [pro-react.github.io](http://pro-react.github.io)。此外，可访问 [www.tupwk.com.cn/downpage](http://www.tupwk.com.cn/downpage)，输入中文书名或中文 ISBN 来下载源代码，也可以扫描本书封底的二维码下载资料。

## 联系作者

感谢你购买本书。我希望能享受阅读本书的乐趣，并从中获取有价值的信息。欢迎你个人针对本书内容与源代码给我提供反馈、问题和评论。你可以通过 [proreactbook@gmail.com](mailto:proreactbook@gmail.com) 联系我。

祝你好运！期待着你的 React 应用程序的诞生！

# 目 录

第 1 章 React 入门	1
1.1 开始学习之前	1
1.1.1 Node.js 和 npm	1
1.1.2 JavaScript ES6	2
1.2 定义 React	2
1.3 React 的优点	2
1.3.1 简单易学的响应式渲染	3
1.3.2 使用纯 JavaScript 进行 面向组件开发	3
1.3.3 灵活的文档模型抽象表现	4
1.4 创建你的第一个 React 应用程序	4
1.4.1 React 开发流程	4
1.4.2 创建你的第一个组件	8
1.4.3 减少输入的字符数量	9
1.4.4 动态值	10
1.5 将组件组合起来	10
1.5.1 props	10
1.5.2 呈现看板应用	11
1.5.3 定义组件的层次关系	13
1.5.4 props 的重要性	14
1.5.5 创建组件	14
1.6 介绍 state	21
1.7 本章小结	23
第 2 章 深入 DOM 抽象	25
2.1 React 中的事件	25
2.1.1 DOM 事件侦听器	25
2.1.2 看板应用：管理 DOM 事件	26
2.2 深入了解 JSX	27

2.2.1 JSX 与 HTML	28
2.2.2 JSX 和 HTML 的 不同之处	28
2.2.3 JSX 的怪异之处	29
2.3 看板应用：指示卡片的 打开和关闭状态	31
2.3.1 空格	32
2.3.2 JSX 中的注释	33
2.3.3 渲染动态 HTML	33
2.3.4 看板应用：渲染 Markdown	33
2.4 脱离 JSX 的 React	36
2.4.1 普通 JavaScript 中的 React 元素	36
2.4.2 元素工厂	36
2.4.3 自定义工厂	37
2.5 内联样式	37
2.5.1 定义内联样式	37
2.5.2 看板应用：通过内联样式 定义卡片颜色	38
2.6 使用表单	40
2.6.1 受控组件	40
2.6.2 特例	42
2.6.3 非受控组件	43
2.6.4 看板应用：创建一个 任务表单	44
2.7 幕后的虚拟 DOM	44
2.7.1 key 属性	45
2.7.2 看板应用：key	45
2.7.3 refs	47
2.8 本章小结	48

第 3 章 使用组件构建应用程序.....49

3.1 校验属性.....49

3.1.1 属性的默认值 .....50

3.1.2 内置的 propTypes 校验器...51

3.1.3 为看板应用定义  
propTypes .....52

3.1.4 自定义 propTypes 校验器...54

3.2 组件组合的策略与  
最佳实践.....55

3.2.1 有状态的组件和单纯组件...55

3.2.2 哪些组件应当是有  
状态组件.....56

3.2.3 数据流和组件通信.....59

3.3 组件的生命周期.....63

3.3.1 声明周期的阶段与函数.....63

3.3.2 生命周期函数实践：  
数据获取 .....64

3.4 浅谈不变性.....67

3.4.1 普通 JavaScript 中的  
不变性 .....67

3.4.2 嵌套对象 .....69

3.4.3 React 不变性助手 .....70

3.5 看板应用：添加一点  
复杂性.....73

3.5.1 从外部 API 获取初始的  
卡片数据.....73

3.5.2 将任务回调以 props 传递...76

3.5.3 处理任务数据 .....80

3.5.4 基本的乐观更新回滚.....83

3.6 本章小结.....87

第 4 章 复杂交互.....89

4.1 React 中的动画.....89

4.1.1 CSS 过渡和动画基础.....89

4.1.2 ReactCSSTransitionGroup...95

4.2 拖放.....100

4.2.1 React DnD 实现概述.....101

4.2.2 React DnD 实现示例 .....101

4.3 看板应用：支持动画和  
拖放 .....113

4.3.1 卡片切换动画.....113

4.3.2 卡片的拖曳 .....115

4.4 本章小结 .....129

第 5 章 路由.....131

5.1 使用原生方式实现路由 .....131

5.2 React Router .....135

5.2.1 Index 路由.....138

5.2.2 带参数的路由.....139

5.2.3 设置活动链接.....144

5.2.4 传递 props.....144

5.2.5 将 UI 界面与 URL 解耦...147

5.2.6 在代码中更改路由 .....149

5.2.7 History 库.....152

5.2.8 看板应用：实现  
路由功能 .....153

5.3 本章小结 .....166

第 6 章 结合 Flux 的 React  
应用程序架构.....167

6.1 什么是 Flux.....167

6.1.1 Store.....167

6.1.2 Action.....168

6.1.3 Dispatcher.....169

6.2 假想的简单 Flux  
应用程序 .....169

6.3 Flux 工具包.....177

6.3.1 Flux Store 工具 .....177

6.3.2 容器组件高阶函数.....180

6.4 异步 Flux.....181

6.4.1 waitFor：协调 Store 的  
更新数序 .....181

6.4.2 异步数据获取.....184

6.5 AirCheap 应用程序.....184

6.5.1 搭建：项目组织和  
基本文件 .....184





# React 入门

React 是由 Facebook 创建的一个开源项目。它提供了一种在 JavaScript 中构建用户界面的全新方式。自从它公开发布后，这个库迅速风靡全球，并且围绕着它培育了一个生机勃勃的社区。

通过阅读本书，你将掌握在项目中使⽤ React 所需的方方面面。因为 React 只关注 UI 界面的渲染，而不会对应用程序的其他模块所使用的技术做任何假设，所以本书同时也将介绍能匹配 React 模式的路由(Routing)和应用程序架构。

在本章中，我们将从一个较高的层面讲述一些主题，以便你能尽快开始创建应用程序。这些主题包括：

- React 的完整定义，以及优点概览
- 如何使用 JSX，这是一个在 React 中用来渲染 UI 的 JavaScript 语法扩展
- 如何创建包含属性和状态的 React 组件

## 1.1 开始学习之前

React 针对的是现代风格的 JavaScript 开发生态系统。为能亲自尝试本书中的代码示例，你需要安装 Node.js 和 npm。此外，还需要熟悉函数式 JavaScript 语法风格和这门语言最新的特性，如箭头函数(arrow functions)和类。

### 1.1.1 Node.js 和 npm

JavaScript 自诞生之日起就运行在浏览器上，但是通过 Node.js 的开源命令行工具，可以使 JavaScript 运行在你的本地计算机和服务器上。与 npm(Node Package Manager)一道，Node.js 已经成为一项在本地计算机上进行 JavaScript 应用程序开发的极有用工具，它使得开发人员可以创建脚本来运行任务(例如，复制和移动文件，或是启动一个本地开发服务器)，以及自动下载应用程序所依赖的组件。

如果你尚未安装 Node.js，现在就下载它的 Windows、Mac 或者 Linux 版本，将其安装到你的计算机上。下载地址为 <https://nodejs.org/>。

### 1.1.2 JavaScript ES6

JavaScript 是一门自诞生起多年一直在不断进化的语言。最近, JavaScript 技术社区已经认同了一组新的语言特性。有一些最新的浏览器已经能够支持这些特性, React 社区也广泛地使用了这些新的特性(例如, 箭头函数、类、展开操作符)。React 同时鼓励在 JavaScript 代码中使用函数式编程模式, 所以你也需要熟悉在 JavaScript 中函数和上下文是如何工作的, 这样你才能了解 `map`、`reduce` 和 `assign` 等方法。如果你对这些细节感觉有些模糊, 可参阅 Apress 网站([www.apress.com/](http://www.apress.com/))和本书的 GitHub 页面(<http://pro-react.github.io/>)上有关这些主题的在线附录说明。

## 1.2 定义 React

为清楚地说明 React 究竟为何物, 我将对它做如下定义:

React 是一个使用 JavaScript 和 XML 技术(可选)构建可组合用户界面的引擎。

下面对 React 定义的每个部分再详加说明:

**React 是一个引擎:** React 的网站将它定义为一个库, 但是我觉得使用“引擎”这个词更能体现出 React 的核心优势: 用来实现响应式 UI 渲染的方式。React 的方式是将状态(在一个给定的时间点, 所有用来定义应用程序的内部数据)从展现给用户的 UI 中分离出来。在 React 中, 你可以声明如何将应用程序的状态表现为 DOM 的虚拟元素, 然后自动更新 DOM 以反映出状态的变化。

“引擎”这个术语首先被 Justin Deal 用来描述 React, 因为他觉得 React 渲染 UI 界面的方式和游戏引擎渲染的工作方式十分相似(<https://zapier.com/engineering/react-js-tutorial-guide-gotchas/>)。

**创建可组合用户界面:** 减少创建和维护用户界面的复杂性一直是 React 的核心目标。React 拥抱了这样一种理念: 将 UI “打散”成易于重用、扩展和维护的组件与自包含的、关注特定用途的构件(building blocks)。

**使用 JavaScript 和 XML 技术(可选):** React 是一个可用于浏览器、服务器和移动设备之上的纯 JavaScript 库。如你将在本章中所见, React 有一种可选的语法来让你可以使用 XML 来描述 UI。一开始你可能会对这种语法感到有些陌生, 但使用 XML 对于描述用户界面其实有诸多优点, 包括: XML 是声明性的, 很容易通过 XML 观察元素之间的关系, 也很容易使 UI 的整体结构可视化。

## 1.3 React 的优点

市面上有许多的 JavaScript MVC 框架。Facebook 有什么理由还要创建 React? 你有什么理由要使用 React? 下面的三节内容将探索 React 的一些优点, 从而回答这两个问题。



### 1.3.1 简单易学的响应式渲染

在 Web 开发的早期,那时还根本没有单页应用程序这个概念,用户每次在页面上进行一次交互(比如单击了一个按钮),就算新的页面只和原有页面仅有一点不同,服务器都会将一整页新的页面发送回客户端浏览器。从用户的角度看,体验会非常糟糕,不过程序员倒是很容易规划出用户在某一时刻能看到哪些内容。

单页应用程序持续地从服务器获取新数据,并在用户进行交互时变换 DOM 上面的部分内容。在用户界面逐渐变得复杂时,应用程序也要实现越来越复杂的逻辑来检验应用程序的当前状态,并对 DOM 及时进行所需的修改。

许多 JavaScript 框架(特别是那些在 React 出现之前的框架)都使用了数据绑定技术,来处理这种日益增加的复杂性并保持用户界面和应用程序状态的同步,但是数据绑定在可维护性、可扩展性和性能上,都有一些缺陷。

响应式渲染比传统的数据绑定技术要更容易使用一些。它让我们用一种声明方式,来定义组件的外观和行为。当数据发生变化时,React 在概念上会重新渲染整个用户界面。

当然,每次在状态数据发生变化时就真的重新渲染整个用户界面,在性能上是不可接受的,React 使用了一种存在于内存中的轻量级 DOM 表示法,这被称为“虚拟 DOM”。

处理这种内存中的虚拟 DOM 要比处理真正的 DOM 更快、更有效。当因为用户的交互或者数据的获取而导致应用程序的状态发生改变时,React 快速地将 UI 的当前状态与期望的状态进行比较,然后计算出要对真实 DOM 所需进行的最小更改。这种工作方式使得 React 非常快、非常高效。即使在一个移动设备上,React 应用程序也能轻松地达到 60fps 的刷新率。

### 1.3.2 使用纯 JavaScript 进行面向组件开发

在一个 React 应用程序中,一切都由组件组成,组件就是应用程序中的自包含的、关注特定用途的基础构件。基于组件来开发应用程序使用了一种“分而治之”的途径,来避免在某个地方有太多的复杂性。由于组件可以组合起来,每个组件都可以尽量保持小巧,通过将更小的组件组合在一起,创建复杂的包含更多功能的组件就变得简单了。

不同于使用特定的模板语言或是传统 Web 应用程序 UI 中用到的 HTML 标记,React 组件由普通的 JavaScript 写就。使用普通 JavaScript 代码编写组件有一个很好的理由:模板限定了你可用来构造 UI 界面时能使用到的功能特性。React 则是使用一门功能完整的编程语言来渲染界面,这相对于使用模板具备很大的优势。

另外,通过使组件成为自包含的、并在相关视图逻辑中使用一个统一的标记,React 组件实现了关注点的分离。在 Web 开发的早期岁月,关注点的分离是通过在不同部分使用不同的语言来强制实现的:内容结构使用 HTML,样式使用 CSS,逻辑行为使用 JavaScript。这种方法在问世之初工作得很好,因为那个时候 Web 页面还是静态呈现的。但是现在用户界面变得更有交互性、更复杂,显示逻辑和 HTML 标记不可避免地被绑定在一起;标记、样式和 JavaScript 之间的分离变成了仅是技术上的分离,而非关注点的分离。

React 假设显示逻辑和 HTML 标记是高度粘合的；它们同时用来实现 UI 的展现，并通过为每个关注点创建离散的、良好封装的、可重用的组件，来鼓励实现关注点的分离。

### 1.3.3 灵活的文档模型抽象表现

React 内置了自己的一个 UI 轻量级表现模型，以抽象出 UI 底层的文档模型。这种方式最值得一提的优点，就是不论在 Web 页面，还是在原生的 iOS 和 Android 界面上，它都可以使用同样的原则来渲染 HTML。这种抽象表现同时带来了下面这些特性：

- 事件在所有浏览器和设备上都会以一种统一、标准的方式，自动地使用代理，来达到行为的一致性。
- 为实现 SEO(搜索引擎优化)和更好的性能，React 组件可在服务器上被渲染。

## 1.4 创建你的第一个 React 应用程序

现在你已经知道了组件是 React UI 的基础构件，但是组件到底看起来是什么样子的？怎样才能创建一个组件呢？简单来说，一个 React 组件就是一个带有 render 方法，并且返回组件 UI 描述的 JavaScript 类，如下所示：

```
class Hello extends React.Component {
  render() {
    return (
      <h1>Hello World</h1>
    )
  }
}
```

你也许已经注意到了 JavaScript 代码中间的 HTML 标记。之前曾经提到过，React 有一种称为 JSX 的 JavaScript 语法扩展，可以让我们在代码中直接书写 XML(以生成 HTML)。

是否使用 JSX 是可选的，但是它已经成为一种被广泛接受的在 React 组件中定义 UI 的标准方案，由于 JSX 使用了具有丰富表达能力的声明式语法，以及这些内容最终会被转换成普通的 JavaScript 函数调用，所以这意味着 JSX 并没有影响 JavaScript 语言原本的语义。

我们会在下一章更详尽地介绍 JSX，但是现在要提醒你注意的是，React 需要一个额外的“转换”步骤(或者说是翻译步骤)，将 JSX 转换成 JavaScript 代码。

在现代的 JavaScript 开发生态系统中，有许多工具可处理这种需要额外转换步骤的情况。下面花一点时间来讨论如何为 React 项目搭建出一套流畅的开发流程。

### 1.4.1 React 开发流程

想当年，我们都是将所有 JavaScript 代码都写到一个文件里面，手工下载一两个 JavaScript 库，然后将自己的代码和第三方的库统统塞到一个页面上。当然，现在你仍然

可将 React 库的压缩 JavaScript 文件通过下载或复制粘贴的方法获得,然后立即用它来运行组件,并在运行时转换 JSX。只不过现在除了小的演示和原型之外,没人会在真正的项目里面这样做。

即使是创建一个最简单的应用程序,我们也希望通过开发流程满足如下需求:

- 编写 JSX 并随时将它转换成标准的 JavaScript 代码
- 使用模块化模式编写代码
- 管理依赖性
- 打包多个 JavaScript 文件并使用 source maps 进行调试

为满足上述需求,一个 React 项目的基础结构包含如下内容:

(1) 一个 source 文件夹,里面包含了你所编写的所有 JavaScript 模块。

(2) 一个 index.html 文件。在 React 应用程序中,这个 HTML 页面通常差不多是一个空页面。它仅用来加载应用程序的 JavaScript,并提供一个 div 元素(或者其他任何元素)来让 React 在该元素中渲染出应用程序的组件。

(3) 一个 package.json 文件。这个 package.json 是一个独立的 npm 清单文件,它包含了有关项目的诸多信息,例如名称、描述、作者的信息等。它让开发人员在其中指定依赖关系(通过指定依赖关系可以实现模块的自动下载和安装)并定义脚本任务。

(4) 一个模块打包或 build 工具,用来实现 JSX 转换和模块/依赖项打包。模块的应用使得 JavaScript 代码被组织到了多个文件中,每个模块都声明了它自己的依赖项。模块打包工具在编写完代码之后,就可以根据依赖关系,以正确顺序将所有代码文件打包在一起。有许多工具都可以实现这样的功能,包括 Grunt、Gulp 和 Brunch 等。在任何上述工具的文档中,你都能找到如何配合 React 使用的内容,不过通常来说,React 社区将 webpack 用作完成这项工作的首选工具。本质上,webpack 是一个模块打包工具,但它也可将源代码通过加载器进行转换和编译。

图 1-1 展现了上面所提到的文件和文件夹结构。

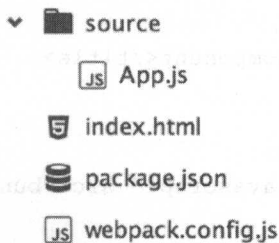


图 1-1 最简单 React 项目的文件和文件夹结构

#### 提示:

在本书的在线资料中,你可以找到一个关于如何使用 webpack 设置一个 React 项目的完整附录说明。附录中涵盖了 webpack 的详细信息,并展现了如何设置高级选项,例如,热重载(Hot Reloading)React 组件。在线附录位于 Apress 网站([www.apress.com](http://www.apress.com))和本书的 GitHub 页面([pro-react.github.io](http://pro-react.github.io))上。

## 1. 快速起步

为专注于学习 React 库，本书提供了一个 React 应用程序模板包。你可从 [apress.com](https://apress.com) 或者 GitHub 页面(<https://github.com/pro-react/react-app-boilerplate>)下载它。模板项目包含了立即开发 React 程序所需的所有基本文件和配置。下载后，你要做的就是安装依赖项并运行开发服务器来在浏览器中测试项目。要自动安装所有依赖项，可打开终端或命令提示符，运行 `npm install`。要运行开发服务器，输入 `npm start`。

你已经准备好了。现在可以直接跳过下面的主题，直接开始创建你的第一个 React 组件了。

## 2. 或者，自己动手

如果想要尝试自己动手，可通过下面的 5 个步骤来手工创建基础的项目结构。由于本书的主要关注点还是 React 库，所以我们现在不会深入讲解每个步骤中的细节，或是讲解那些可选的配置项，但你可通过在线附录阅读到有关它们的更多内容，或是查看 React 应用程序模板项目的源码文件。在线附录和模板项目都可从 Apress 网站([www.apress.com/](http://www.apress.com/))或是本书的 GitHub 页面(<http://pro-react.github.io/>)下载。

(1) 首先创建 source 文件夹(source 和 app 是常用的文件夹名字)。这个文件夹将只包含 JavaScript 模块。不需要被模块打包工具处理的静态资源(包括 `index.html`、图片文件，CSS 文件等)将放置到根文件夹中。

(2) 在项目的根文件夹中创建 `index.html` 文件。文件内容如代码清单 1-1 所示。

**代码清单 1-1：加载打包后的 JavaScript 代码并为 React 组件渲染提供根 Div 的简单 HTML 页面**

```
<!DOCTYPE html>
<html>
  <head>
    <title>First React Component</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript" src="bundle.js"></script>
  </body>
</html>
```

(3) 通过在终端或者命令提示符上运行 `npm init`，然后按照提示进行操作来创建 `package.json` 文件。你将使用 `npm` 来进行依赖项管理(下载和安装所有需要的库)。你的项目的依赖项包括了 React、进行 JSX 转换的 Babel 编译器(loader 和 core)以及 webpack(包括 webpack 开发服务器)。按照代码清单 1-2 所示编辑你的 `package.json` 文件，然后运行 `npm install`。

代码清单 1-2: 一个包含了依赖项的简单 package.json 文件

```
{
  "name": "your-app-name",
  "version": "X.X.X",
  "description": "Your app description",
  "author": "You",
  "devDependencies": {
    "babel-core": "^5.8.*",
    "babel-loader": "^5.3.*",
    "webpack": "^1.12.*",
    "webpack-dev-server": "^1.10.*"
  },
  "dependencies": {
    "react": "^0.13.*"
  }
}
```

(4) 接下来需要配置 webpack, 它是你所选择的模块打包工具。代码清单 1-3 显示了配置文件。让我们来看一下配置文件里面的内容。首先, entry 键指向了应用程序主模块。

代码清单 1-3: webpack.config.js 文件

```
module.exports = {
  entry: [
    './source/App.js'
  ],
  output: {
    path: __dirname,
    filename: "bundle.js"
  },
  module: {
    loaders: [{
      test: /\.jsx?$/,
      loader: 'babel'
    }]
  }
};
```

下一个键 output 告诉 webpack 将按正确顺序把所有模块打包后的单个 JavaScript 文件保存到什么地方。

最后, 在 module 键的 loaders 区域, 将所有 .js 文件传送给 Babel, 这个 JavaScript 编译器会将所有 JSX 转换成标准的 JavaScript 代码。记住, Babel 能做的远不止于此, 它可以让你的代码使用最新的 JavaScript 语法, 例如箭头函数和类。

(5) 现在是最后一个步骤。项目的结构已经完整。启动一个本地服务器(用于在浏览器中进行测试)的命令是“node\_modules/.bin/webpack-dev-server”, 不过为避免每次都要手工输入这么长一段命令, 你可编辑步骤(3)中创建的 package.json 文件, 将这个长命令



定义为一个任务，如代码清单 1-4 所示。

代码清单 1-4：将启动脚本添加到 package.json 文件中

```
{
  "name": "your-app-name",
  "version": "X.X.X",
  "description": "Your app description",
  "author": "You",
  "scripts": {
    "start": "node_modules/.bin/webpack-dev-server --progress"
  },
  "devDependencies": {
    "babel-core": "^5.8.*",
    "babel-loader": "^5.3.*",
    "webpack": "^1.12.*",
    "webpack-dev-server": "^1.10.*"
  },
  "dependencies": {
    "react": "^0.13.*"
  }
}
```

完成这个步骤后，下次想要运行本地的开发服务器时，只需要输入 `npm start`。

### 1.4.2 创建你的第一个组件

现在你已经有了一个基本的项目结构，它帮我们管理依赖项，提供了一个模块化系统，并为你转换 JSX。现在可以开始创建一个 Hello World 组件，并将它渲染到页面上了。你将继续使用本章之前展示过的组件的代码，不过在顶部加上了一个 `import` 声明，这个声明使 React 库被打包到最终要执行的 JavaScript 文件中。

```
import React from 'react';

class Hello extends React.Component {
  render() {
    return (
      <h1>Hello World</h1>
    );
  }
}
```

接下来，将使用 `React.render` 在页面上显示组件，如下面一行代码和图 1-2 所示：

```
React.render(<Hello />, document.getElementById('root'));
```



图 1-2 在浏览器中显示的第一个组件

**提示:**

可在一个页面的文档 body 中直接渲染组件的内容,但是通常来说,在一个子元素(通常是一个 div 元素)中进行渲染要更好一些。许多 JavaScript 库和浏览器扩展都将节点附加到文档 body 中,由于 React 需要完全掌控 DOM 树,如果 React 直接使用文档 body 而非一个子元素进行渲染,第三方库和浏览器扩展的这些行为就会导致不可预见的问题。

### 1.4.3 减少输入的字符数量

许多开发人员都经常使用一个技巧来减少输入的字符数量,那就是在模块的 import 中使用解构赋值来直接访问模块内部的函数和类。在上面的那个代码示例中,我们可使用下面的这个方法避免需要输入完整的“React.Component”:

```
import React, { Component } from 'react';

class Hello extends Component {
  render() {
    return (
      <h1>Hello World</h1>
    );
  }
}
```

在这里,示例中看起来也没节省太多需要输入的字符,但是在更大项目中,累积起来所节省的字符就比较可观了。

**注意:**

解构赋值是属于下个版本 JavaScript 规范的一个特性。在 React 中可以使用这种未来 JavaScript 版本才有的特性,在线附录 C 对此进行了详细描述。

### 1.4.4 动态值

在 JSX 中, 位于花括号({})中的值会被当作一个 JavaScript 表达式进行求值, 并被渲染到 HTML 标记中。例如, 如果想要显示一个本地变量的值, 可以这样做:

```
import React, { Component } from 'react';

class Hello extends Component {
  render() {
    var place = "World";
    return (
      <h1>Hello {place}</h1>
    );
  }
}

React.render(<Hello />, document.getElementById("root"));
```

## 1.5 将组件组合起来

React 鼓励开发人员尽量创建简单且可重用的组件, 然后将组件进行嵌套和组合来创建复杂的 UI。现在你已经了解了一个 React 组件的基本结构, 下面介绍如何将组件组合在一起。

### 1.5.1 props

要让组件可以重用和组合, 关键是对它们进行配置, React 提供了属性(简称为 props)来实现组件的配置。props 是 React 中的一种从父组件向子组件传输数据的机制。它们在子组件里面不能被修改; props 由父组件传输出去, 也被父组件所“拥有”。

在 JSX 中, props 就像是 HTML 的标签特性那样。作为例子, 下面创建一个简单的商店物品列表, 它将由两个组件组成, 父组件 GroceryList 和子组件 GroceryItem:

```
import React, { Component } from 'react';

// Parent Component
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity="1" name="Bread" />
        <ListItem quantity="6" name="Eggs" />
        <ListItem quantity="2" name="Milk" />
      </ul>
    );
  }
}
```



```
// Child Component
class ListItem extends Component {
  render() {
    return (
      <li>
        {this.props.quantity}× {this.props.name}
      </li>
    );
  }
}

React.render(<GroceryList />, document.getElementById("root"));
```

除了使用命名 props 外，还可以通过 props.children 来引用位于前置标签和后置标签之间的内容：

```
import React, { Component } from 'react';

// Parent Component
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity="1">Bread</ListItem>
        <ListItem quantity="6">Eggs</ListItem>
        <ListItem quantity="2">Milk</ListItem>
      </ul>
    );
  }
}

// Child Component
class ListItem extends Component {
  render() {
    return (
      <li>
        {this.props.quantity}× {this.props.children}
      </li>
    );
  }
}

React.render(<GroceryList />, document.getElementById('root'));
```

## 1.5.2 呈现看板应用

在本书中，针对每个主题，我们都会创建多个小组件和示例代码。我们还要创建一

个完整的应用程序：一个看板风格的项目管理工具。

在一个看板上，项目活动以卡片形式展现(如图 1-3 所示)。卡片根据它们的状态被归类到不同列表中，卡片可从一个列表移到另一个列表中，以表现一个功能从计划到实现的流程。

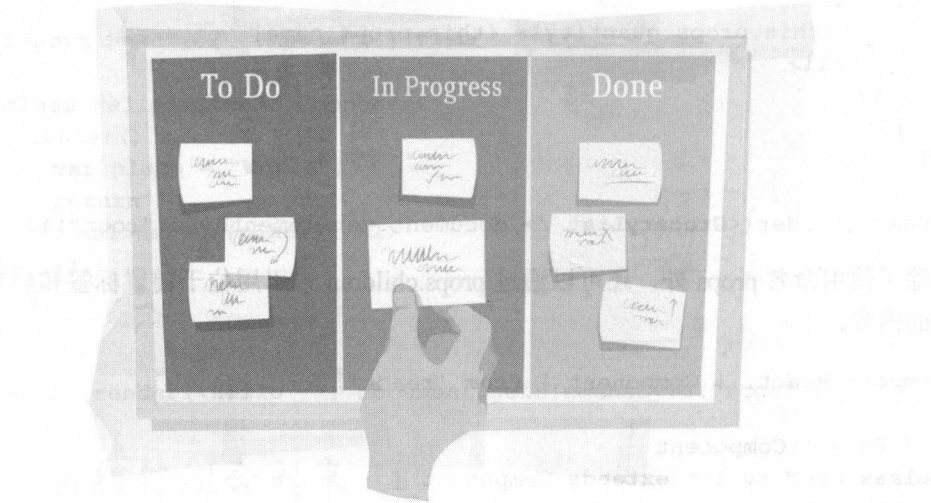


图 1-3 一个看板示例

网上有很多现成的看板风格项目管理应用程序。Trello.com 就是这种类型的一个著名网站，虽说你的项目通常没有复杂到要使用 Trello.com 的程度。你的项目最终看起来会如图 1-4 所示，看板应用的数据模型如代码清单 1-5 所示。

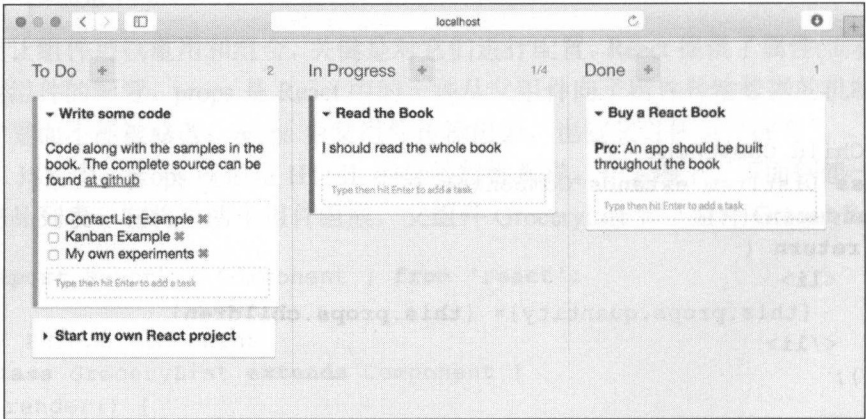


图 1-4 在后续章节中将要创建的看板应用

代码清单 1-5: 看板应用的数据模型

```
[
  {
    id: 1,
    title: "Card one title",
    description: "Card detailed description.",
  },
  {
    id: 2,
    title: "Card two title",
    description: "Card detailed description.",
  },
  {
    id: 3,
    title: "Card three title",
    description: "Card detailed description.",
  },
  {
    id: 4,
    title: "Card four title",
    description: "Card detailed description.",
  },
  {
    id: 5,
    title: "Card five title",
    description: "Card detailed description.",
  },
  {
    id: 6,
    title: "Card six title",
    description: "Card detailed description.",
  },
  {
    id: 7,
    title: "Card seven title",
    description: "Card detailed description.",
  },
  {
    id: 8,
    title: "Card eight title",
    description: "Card detailed description.",
  },
  {
    id: 9,
    title: "Card nine title",
    description: "Card detailed description.",
  },
  {
    id: 10,
    title: "Card ten title",
    description: "Card detailed description.",
  },
  {
    id: 11,
    title: "Card eleven title",
    description: "Card detailed description.",
  },
  {
    id: 12,
    title: "Card twelve title",
    description: "Card detailed description.",
  },
  {
    id: 13,
    title: "Card thirteen title",
    description: "Card detailed description.",
  },
  {
    id: 14,
    title: "Card fourteen title",
    description: "Card detailed description.",
  },
  {
    id: 15,
    title: "Card fifteen title",
    description: "Card detailed description.",
  },
  {
    id: 16,
    title: "Card sixteen title",
    description: "Card detailed description.",
  },
  {
    id: 17,
    title: "Card seventeen title",
    description: "Card detailed description.",
  },
  {
    id: 18,
    title: "Card eighteen title",
    description: "Card detailed description.",
  },
  {
    id: 19,
    title: "Card nineteen title",
    description: "Card detailed description.",
  },
  {
    id: 20,
    title: "Card twenty title",
    description: "Card detailed description.",
  },
  {
    id: 21,
    title: "Card twenty-one title",
    description: "Card detailed description.",
  },
  {
    id: 22,
    title: "Card twenty-two title",
    description: "Card detailed description.",
  },
  {
    id: 23,
    title: "Card twenty-three title",
    description: "Card detailed description.",
  },
  {
    id: 24,
    title: "Card twenty-four title",
    description: "Card detailed description.",
  },
  {
    id: 25,
    title: "Card twenty-five title",
    description: "Card detailed description.",
  },
  {
    id: 26,
    title: "Card twenty-six title",
    description: "Card detailed description.",
  },
  {
    id: 27,
    title: "Card twenty-seven title",
    description: "Card detailed description.",
  },
  {
    id: 28,
    title: "Card twenty-eight title",
    description: "Card detailed description.",
  },
  {
    id: 29,
    title: "Card twenty-nine title",
    description: "Card detailed description.",
  },
  {
    id: 30,
    title: "Card thirty title",
    description: "Card detailed description.",
  },
  {
    id: 31,
    title: "Card thirty-one title",
    description: "Card detailed description.",
  },
  {
    id: 32,
    title: "Card thirty-two title",
    description: "Card detailed description.",
  },
  {
    id: 33,
    title: "Card thirty-three title",
    description: "Card detailed description.",
  },
  {
    id: 34,
    title: "Card thirty-four title",
    description: "Card detailed description.",
  },
  {
    id: 35,
    title: "Card thirty-five title",
    description: "Card detailed description.",
  },
  {
    id: 36,
    title: "Card thirty-six title",
    description: "Card detailed description.",
  },
  {
    id: 37,
    title: "Card thirty-seven title",
    description: "Card detailed description.",
  },
  {
    id: 38,
    title: "Card thirty-eight title",
    description: "Card detailed description.",
  },
  {
    id: 39,
    title: "Card thirty-nine title",
    description: "Card detailed description.",
  },
  {
    id: 40,
    title: "Card forty title",
    description: "Card detailed description.",
  },
  {
    id: 41,
    title: "Card forty-one title",
    description: "Card detailed description.",
  },
  {
    id: 42,
    title: "Card forty-two title",
    description: "Card detailed description.",
  },
  {
    id: 43,
    title: "Card forty-three title",
    description: "Card detailed description.",
  },
  {
    id: 44,
    title: "Card forty-four title",
    description: "Card detailed description.",
  },
  {
    id: 45,
    title: "Card forty-five title",
    description: "Card detailed description.",
  },
  {
    id: 46,
    title: "Card forty-six title",
    description: "Card detailed description.",
  },
  {
    id: 47,
    title: "Card forty-seven title",
    description: "Card detailed description.",
  },
  {
    id: 48,
    title: "Card forty-eight title",
    description: "Card detailed description.",
  },
  {
    id: 49,
    title: "Card forty-nine title",
    description: "Card detailed description.",
  },
  {
    id: 50,
    title: "Card fifty title",
    description: "Card detailed description.",
  },
  {
    id: 51,
    title: "Card fifty-one title",
    description: "Card detailed description.",
  },
  {
    id: 52,
    title: "Card fifty-two title",
    description: "Card detailed description.",
  },
  {
    id: 53,
    title: "Card fifty-three title",
    description: "Card detailed description.",
  },
  {
    id: 54,
    title: "Card fifty-four title",
    description: "Card detailed description.",
  },
  {
    id: 55,
    title: "Card fifty-five title",
    description: "Card detailed description.",
  },
  {
    id: 56,
    title: "Card fifty-six title",
    description: "Card detailed description.",
  },
  {
    id: 57,
    title: "Card fifty-seven title",
    description: "Card detailed description.",
  },
  {
    id: 58,
    title: "Card fifty-eight title",
    description: "Card detailed description.",
  },
  {
    id: 59,
    title: "Card fifty-nine title",
    description: "Card detailed description.",
  },
  {
    id: 60,
    title: "Card sixty title",
    description: "Card detailed description.",
  },
  {
    id: 61,
    title: "Card sixty-one title",
    description: "Card detailed description.",
  },
  {
    id: 62,
    title: "Card sixty-two title",
    description: "Card detailed description.",
  },
  {
    id: 63,
    title: "Card sixty-three title",
    description: "Card detailed description.",
  },
  {
    id: 64,
    title: "Card sixty-four title",
    description: "Card detailed description.",
  },
  {
    id: 65,
    title: "Card sixty-five title",
    description: "Card detailed description.",
  },
  {
    id: 66,
    title: "Card sixty-six title",
    description: "Card detailed description.",
  },
  {
    id: 67,
    title: "Card sixty-seven title",
    description: "Card detailed description.",
  },
  {
    id: 68,
    title: "Card sixty-eight title",
    description: "Card detailed description.",
  },
  {
    id: 69,
    title: "Card sixty-nine title",
    description: "Card detailed description.",
  },
  {
    id: 70,
    title: "Card seventy title",
    description: "Card detailed description.",
  },
  {
    id: 71,
    title: "Card seventy-one title",
    description: "Card detailed description.",
  },
  {
    id: 72,
    title: "Card seventy-two title",
    description: "Card detailed description.",
  },
  {
    id: 73,
    title: "Card seventy-three title",
    description: "Card detailed description.",
  },
  {
    id: 74,
    title: "Card seventy-four title",
    description: "Card detailed description.",
  },
  {
    id: 75,
    title: "Card seventy-five title",
    description: "Card detailed description.",
  },
  {
    id: 76,
    title: "Card seventy-six title",
    description: "Card detailed description.",
  },
  {
    id: 77,
    title: "Card seventy-seven title",
    description: "Card detailed description.",
  },
  {
    id: 78,
    title: "Card seventy-eight title",
    description: "Card detailed description.",
  },
  {
    id: 79,
    title: "Card seventy-nine title",
    description: "Card detailed description.",
  },
  {
    id: 80,
    title: "Card eighty title",
    description: "Card detailed description.",
  },
  {
    id: 81,
    title: "Card eighty-one title",
    description: "Card detailed description.",
  },
  {
    id: 82,
    title: "Card eighty-two title",
    description: "Card detailed description.",
  },
  {
    id: 83,
    title: "Card eighty-three title",
    description: "Card detailed description.",
  },
  {
    id: 84,
    title: "Card eighty-four title",
    description: "Card detailed description.",
  },
  {
    id: 85,
    title: "Card eighty-five title",
    description: "Card detailed description.",
  },
  {
    id: 86,
    title: "Card eighty-six title",
    description: "Card detailed description.",
  },
  {
    id: 87,
    title: "Card eighty-seven title",
    description: "Card detailed description.",
  },
  {
    id: 88,
    title: "Card eighty-eight title",
    description: "Card detailed description.",
  },
  {
    id: 89,
    title: "Card eighty-nine title",
    description: "Card detailed description.",
  },
  {
    id: 90,
    title: "Card ninety title",
    description: "Card detailed description.",
  },
  {
    id: 91,
    title: "Card ninety-one title",
    description: "Card detailed description.",
  },
  {
    id: 92,
    title: "Card ninety-two title",
    description: "Card detailed description.",
  },
  {
    id: 93,
    title: "Card ninety-three title",
    description: "Card detailed description.",
  },
  {
    id: 94,
    title: "Card ninety-four title",
    description: "Card detailed description.",
  },
  {
    id: 95,
    title: "Card ninety-five title",
    description: "Card detailed description.",
  },
  {
    id: 96,
    title: "Card ninety-six title",
    description: "Card detailed description.",
  },
  {
    id: 97,
    title: "Card ninety-seven title",
    description: "Card detailed description.",
  },
  {
    id: 98,
    title: "Card ninety-eight title",
    description: "Card detailed description.",
  },
  {
    id: 99,
    title: "Card ninety-nine title",
    description: "Card detailed description.",
  },
  {
    id: 100,
    title: "Card one hundred title",
    description: "Card detailed description.",
  },
]
```

```
status: "todo",
tasks: [
  {id: 1, name:"Task one", done:true},
  {id: 2, name:"Task two", done:false},
  {id: 3, name:"Task three", done:false}
],
{ id:2,
  title: "Card Two title",
  description: "Card detailed description",
  status: "in-progress",
  tasks: []
},
{ id:3,
  title: "Card Three title",
  description: "Card detailed description",
  status: "done",
  tasks: []
},
];
```

### 1.5.3 定义组件的层次关系

第一件要做的事情，就是将页面打散成嵌套的组件。有 3 个要注意的事项。

(1) 记住每个组件都应当小巧且只关注单个功能。换句话说，一个组件只应当完成一件事。如果一个组件不断膨胀，就应当将它打散成多个更小的子组件。

(2) 分析项目的外观框架和布局，这些信息通常能提示我们如何决定组件的层次关系。

(3) 看一看应用程序的数据模型。界面和数据模型倾向于体现相同的信息架构，所以将 UI 界面分割成组件通常是很直观的。被打散的组件应该正好能体现数据模型中的一小块。

如果将上面说的概念应用到看板应用，将推导出如图 1-5 所示的组件层次关系。

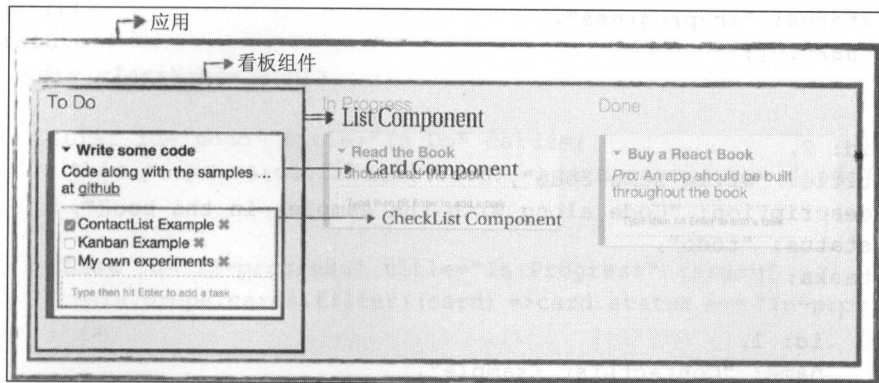


图 1-5 看板应用中的组件层次关系

### 1.5.4 props 的重要性

props 是实现组件组合的关键。props 是 React 中的一种从父组件向子组件传输数据的机制。它们在子组件中不能被修改；props 由父组件传输出去，也被父组件所“拥有”。

### 1.5.5 创建组件

确定了界面上的层级关系之后，就该创建组件了。创建组件有两条主要的途径：从上至下或是从下至上。也就是说，既可以首先创建在层级关系上更高的组件(例如 App 组件)，也可以首先创建在层次关系上更低的组件(例如 CheckList 组件)。为能清楚地理顺从父组件向下传递给子组件的 props 数据，以及子组件如何使用它们，将使用从上至下的方式来创建看板应用的组件。

另外，为让项目的文件结构更具有组织性，更容易维护和实现新功能，将为每个组件创建一个单独的 JavaScript 代码文件。

#### 1. App 模块(app.js)

现在你要尽量让 app.js 文件保持简单。它只会包含数据并只会渲染出一个 KanbanBoard 组件。在看板应用的第一个迭代中，数据将被硬编码到一个本地变量中，但在后续章节，你会从一个 API 中获取数据，代码如代码清单 1-6 所示。

#### 代码清单 1-6：一个简单的 app.js 文件

```
import React from 'react';
import KanbanBoard from './KanbanBoard';

let cardsList = [
  {
    id: 1,
    title: "Read the Book",
    description: "I should read the whole book",
    status: "in-progress",
    tasks: []
  },
  {
    id: 2,
    title: "Write some code",
    description: "Code along with the samples in the book",
    status: "todo",
    tasks: [
      {
        id: 1,
        name: "ContactList Example",
        done: true
      }
    ]
  },
]
```

```

    {
      id: 2,
      name: "Kanban Example",
      done: false
    },
    {
      id: 3,
      name: "My own experiments",
      done: false
    }
  ]
},
];

React.render(<KanbanBoard cards={cardsList} />, document.getElementById(
  'root'));
```

## 2. KanbanBoard 组件(KanbanBoard.js)

KanbanBoard 组件将通过 props 接收数据,并负责筛选状态以创建 3 个 List 组件:“To Do”、“In Progress”和“Done”。代码如代码清单 1-7 所示。

### 注意:

在本章开头曾经说过,React 的组件是使用普通的 JavaScript 代码编写的。它们没有诸如 Mustache 的模板库所具有的分支辅助器之上的对循环的支持,但是由于可以直接使用一门功能完备的编程语言,所以这并不算是一个坏消息。在下面的组件中,会对 cards 数组使用 filter 和 map 函数来对数据进行处理。

### 代码清单 1-7: KanbanBoard 组件

```

import React, { Component } from 'react';
import List from './List';

class KanbanBoard extends Component {
  render() {
    return (
      <div className="app">

        <List id='todo' title="To Do" cards={
          this.props.cards.filter((card) =>card.status === "todo")
        } />

        <List id='in-progress' title="In Progress" cards={
          this.props.cards.filter((card) =>card.status === "in-progress")
        } />

        <List id='done' title='Done' cards={
          this.props.cards.filter((card) =>card.status === "done")
```

```

    } />

    </div>
  );
}
}

```

```
export default KanbanBoard;
```

### 3. List 组件(List.js)

List 组件只会显示出列表名，然后在组件中渲染出所有 Card 组件。注意，你将通过对 props 接收到的 cards 数组进行映射，然后将映射出的标题和描述等单独信息，以 props 方式传递给 Card 子组件。代码如代码清单 1-8 所示。

#### 代码清单 1-8: List 组件

```

import React, { Component } from 'react';
import Card from './Card';

class List extends Component {
  render() {
    var cards = this.props.cards.map((card) => {
      return <Card id={card.id}
        title={card.title}
        description={card.description}
        tasks={card.tasks} />
    });

    return (
      <div className="list">
        <h1>{this.props.title}</h1>
        {cards}
      </div>
    );
  }
}

```

```
export default List;
```

### 4. Card 组件(Card.js)

Card 组件是与用户交互最多的组件，它用来显示一个卡片。每个卡片都有一个标题、一个说明和一个代码清单，如图 1-6 所示，代码如代码清单 1-9 所示。

#### 代码清单 1-9: Card 组件

```

import React, { Component } from 'react';
import CheckList from './CheckList';

```

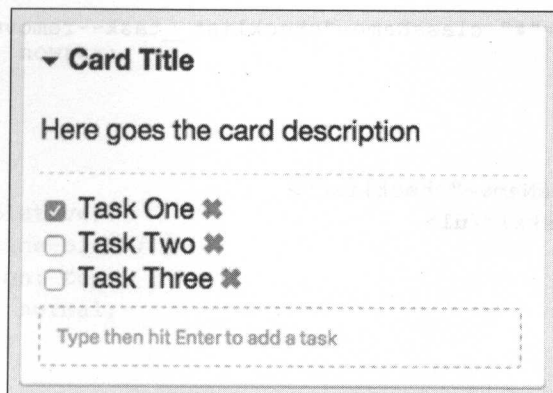


图 1-6 看板应用中的卡片

```
class Card extends Component {
  render() {
    return (
      <div className="card">
        <div className="card_title">{this.props.title}</div>
        <div className="card_details">
          {this.props.description}
          <CheckList cardId={this.props.id} tasks={this.props.tasks} />
        </div>
      </div>
    );
  }
}

export default Card;
```

注意 Card 组件中 `className` 特性的用法。既然 JSX 就是 JavaScript，不鼓励使用类似 `class` 这样的标识符作为 XML 特性，所以在这里使用了 `className`。下一章将对该主题再详加讨论。

## 5. Checklist 组件(CheckList.js)

最后，Checklist 组件用来显示卡片的底部区域。注意，仍然没有一个用来创建新任务的表单，将在之后再创建新建任务的界面。代码如代码清单 1-10 所示。

### 代码清单 1-10: Checklist 组件

```
import React, { Component } from 'react';

class CheckList extends Component {
  render() {
    let tasks = this.props.tasks.map((task) => (
      <li className="checklist_task">
        <input type="checkbox" defaultChecked={task.done} />
        {task.name}
      </li>
    ));
```



```

      <a href="#" className="checklist__task--remove" />
    </li>
  ));

  return (
    <div className="checklist">
      <ul>{tasks}</ul>
    </div>
  );
}
}

export default CheckList;

```

## 6. 界面装饰

React 组件的任务已完成。为让界面美观一点,现在编写一个 CSS 来装饰一下界面(如代码清单 1-11 所示)。别忘了创建一个 HTML 文件来加载 JavaScript 和 CSS 文件,并为 React 放置一个 div 元素来渲染所有组件(代码清单 1-12 展示了一个示例)。

### 代码清单 1-11: CSS 文件

```

*{
  box-sizing: border-box;
}

html,body,#app {
  height:100%;
  margin: 0;
  padding: 0;
}

body {
  background: #eee;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}

h1{
  font-weight: 200;
  color: #3b414c;
  font-size: 20px;
}

ul {
  list-style-type: none;
  padding: 0;
  margin: 0;
}

```



```

.app {
  white-space: nowrap;
  height: 100%;
}

.list {
  position: relative;
  display: inline-block;
  vertical-align: top;
  white-space: normal;
  height: 100%;
  width: 33%;
  padding: 0 20px;
  overflow: auto;
}

.list:not(:last-child):after{
  content: "";
  position: absolute;
  top: 0;
  right: 0;
  width: 1px;
  height: 99%;
  background: linear-gradient(to bottom, #eee 0%, #ccc 50%, #eee 100%) fixed;
}

.card {
  position: relative;
  z-index: 1;
  background: #fff;
  width: 100%;
  padding: 10px 10px 10px 15px;
  margin: 0 0 10px 0;
  overflow: auto;
  border: 1px solid #e5e5df;
  border-radius: 3px;
  box-shadow: 0 1px 0 rgba(0, 0, 0, 0.25);
}

.card__title {
  font-weight: bold;
  border-bottom: solid 5px transparent;
}

.card__title:before {
  display: inline-block;
  width: 1em;
  content: '▶';
}

```

```
.card__title--is-open:before {
  content: '▼';
}

.checklist__task:first-child {
  margin-top: 10px;
  padding-top: 10px;
  border-top: dashed 1px #ddd;
}

.checklist__task--remove:after{
  display: inline-block;
  color: #d66;
  content: "+";
}
```

代码清单 1-12: HTML 文件

```
<!DOCTYPE html>
<html>
<head>
  <title>Kanban App</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div id="root"></div>
  <script type="text/javascript" src="bundle.js"></script>
</body>
</html>
```

如果你在自己的开发环境中一直跟做了上面所有的步骤，那么应该会看到如图 1-7 所示的页面。

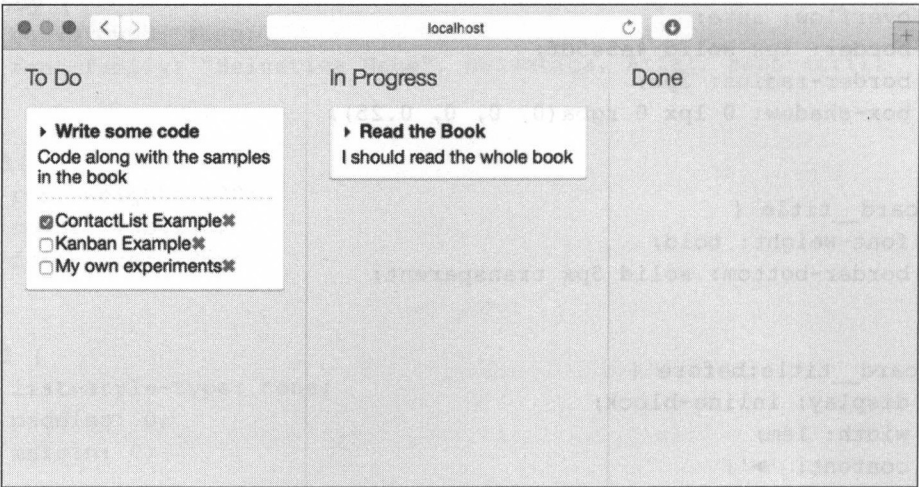


图 1-7 组合的组件界面

## 1.6 介绍 state

至此，你已经看到了 props 的用法，它被组件所接收，且是不可变的。这种不可变的特点导致了组件是静态的。如果想要添加行为和交互，组件就需要有可变的数据来体现它的状态(state)。React 的组件可以在 `this.state` 里面保存可变的数据。注意，`this.state` 对于组件来说是私有的，可通过调用 `this.setState()` 函数来修改它的值。

这时就会引出 React 组件的一个重要特性：当 state 被修改时，组件会触发响应式渲染，组件自身及其子组件都会被重新渲染。如前所述，由于 React 使用了一个虚拟的 DOM，这种重新渲染的操作执行得非常快。

### 看板应用：可切换式卡片

为演示组件中 state 的用法，让我们为看板应用添加一个新功能。你将要使卡片变得可切换。用户可以显示或隐藏卡片的详情。

你可在任何时候为组件设置一个新的 state，但是如果你想要组件有一个初始的 state，你可在类的构造函数里进行设置。现在，Card 组件还没有一个构造函数，只有一个 render 方法。让我们添加一个构造函数，并在组件的 state 中定义一个新的名为 showDetails 的键(key)。代码如代码清单 1-13 所示(为简洁起见，代码清单中省略了 import/export 声明和 render 方法的内容)。

代码清单 1-13：可切换式卡片

```
class Card extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      showDetails: false
    };
  }

  render() {
    return ( ... );
  }
}
```

接下来，可以修改 render 方法中的 JSX 代码，使它只有在 state 的 showDetails 属性为 true 时才渲染卡片的详情。为实现这个功能，定义一个名为 cardDetails 的局部变量，然后只有在当前状态 showDetails 为 true 时才给它赋予真正的数据。在 return 语句中，仅返回这个局部变量的值即可(如果 showDetails 为 false，则变量的值为空)。代码如代码清单 1-14 所示。

代码清单 1-14: Card 组件的 render 方法

```

render() {
  let cardDetails;
  if (this.state.showDetails) {
    cardDetails = (
      <div className="card_details">
        {this.props.description}
        <CheckList cardId={this.props.id} tasks={this.props.tasks} />
      </div>
    );
  };

  return (
    <div className="card">
      <div className="card_title">{this.props.title}</div>
      {cardDetails}
    </div>
  );
}

```

最后一步，让我们添加一个 click 事件处理程序来修改内部 state。使用 JavaScript 的!(非)操作符来切换 showDetails 布尔属性的值(如果它的当前值为 true，应用!操作符后值将是 false，反之亦然)，如代码清单 1-15 所示。

代码清单 1-15: click 事件处理程序

```

render() {
  let cardDetails;
  if (this.state.showDetails) {
    cardDetails = (
      <div className="card_details">
        {this.props.description}
        <CheckList cardId={this.props.id} tasks={this.props.tasks} />
      </div>
    );
  };

  return (
    <div className="card">
      <div className="card_title" onClick={
        ()=>this.setState({showDetails: !this.state.showDetails})
      }>{this.props.title}</div>
      {cardDetails}
    </div>
  );
}

```

当在浏览器上运行这个示例时，卡片上的所有详情默认都会处于关闭状态，通过单

击卡片标题可显示出详情(如图 1-8 所示)。

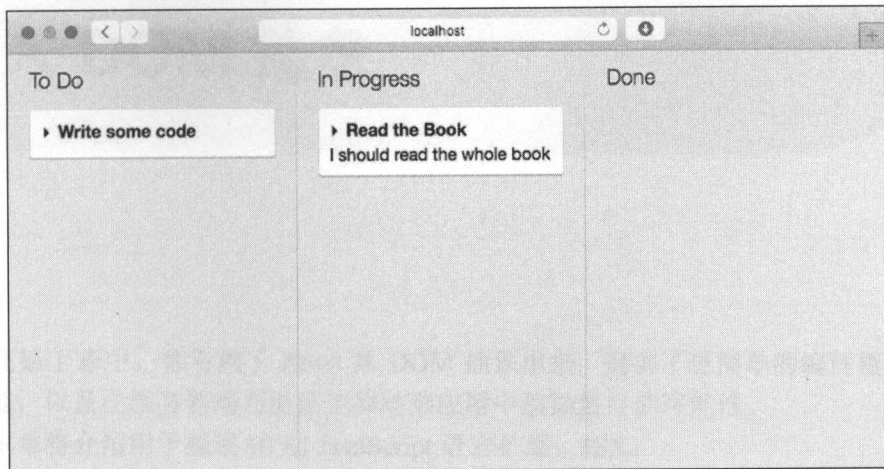


图 1-8 可切换的看板卡片

## 1.7 本章小结

本章解释了 React 的含义以及它给 Web 开发领域带来的好处(主要好处就是提供了一种高性能、声明式途径,能以组件方式构建应用程序的用户界面)。你还创建了你的第一个组件,并见证了 React 组件的所有基本概念:render 方法、JSX、props 和 state。

# 深入 DOM 抽象

在第 1 章中，你看到了 React 从 DOM 抽象出来，提供了更简单的编程模型和更好的性能，以及在服务器端乃至原生移动端应用中渲染组件的可能性。

本章将介绍用于描述 UI 的 JavaScript 语言扩展：JSX。

## 2.1 React 中的事件

React 实现了一个合成事件系统，并为 React 应用程序和界面带来了一致性和高性能。

它通过标准化事件来实现一致性，使得事件在不同的浏览器和平台间都能拥有相同的属性。

它通过自动使用事件委托来实现高性能。React 不会将事件处理程序附加到节点自身。相反，它会将一个单独事件侦听器附加到文档的根节点；当事件被触发后，React 将它映射到适当的组件元素。当组件被卸载时，React 还会自动移除对应的事件侦听器。

### 2.1.1 DOM 事件侦听器

HTML 很早就为标签特性提供了既美观又易于理解的事件处理 API: `onclick`、`onfocus` 等。这种 API 的问题(以及为什么不要在专业性项目中使用它的原因)是它充满了令人讨厌的副作用，简单列举几个：它会污染全局作用域、难以追踪大 HTML 文件的上下文、可能会很慢而且可能导致内存泄漏。

JSX 使用了相似的 API，同样易于使用和理解，但去除了 HTML 带来的令人讨厌的副作用。回调函数的作用域是组件(正如你之前看到过的，组件只负责 UI 的一小部分，并且倾向于包含小段标记)，它会智能地使用事件委托并自动管理卸载。不过请注意，与 HTML 的原生实现相比仍有一些细微的差别。在 React 中，属性采用了驼峰式大小写规则(“`onClick`”而非“`onclick`”)。为建立跨越浏览器和设备的一致性，它实现了各种浏览器的各种版本中的所有变体的一个子集。表 2-1 至表 2-4 展示了可用的事件。



表 2-1 触摸和鼠标事件

onTouchStart	onTouchMove	onTouchEnd	onTouchCancel	
onClick	onDoubleClick	onMouseDown	onMouseUp	onMouseOver
onMouseMove	onMouseEnter	onMouseLeave	onMouseOut	onContextMenu
onDrag	onDragEnter	onDragLeave	onDragExit	onDragStart
onDragEnd	onDragOver	onDrop		

表 2-2 键盘事件

onKeyDown	onKeyUp	onKeyPress
-----------	---------	------------

表 2-3 焦点和表单事件

onFocus	onBlur	
onChange	onInput	onSubmit

表 2-4 其他事件

onScroll	onWheel	onCopy	onCut	onPaste
----------	---------	--------	-------	---------

2.1.2 看板应用：管理 DOM 事件

在看板应用的上个迭代中，你在 `onClick` 事件处理程序中添加了下面的内联函数(使用了胖箭头=>)：

```
<div className="card_title" onClick={
  ()=>this.setState({showDetails: !this.state.showDetails})
}>
```

这样做很实用，却不够灵活。让我们来修改这一实现，在类的内部使用被称为 `toggleDetails` 的新方法来处理事件：

```
class Card extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      showDetails: false
    };
  }

  toggleDetails() {
    this.setState({showDetails: !this.state.showDetails});
  }

  render() {
    let cardDetails;
    if (this.state.showDetails) {
```

```

cardDetails = (
  <div className="card_details">
    {this.props.description}
    <CheckList cardId={this.props.id} tasks={this.props.tasks} />
  </div>
);
}

return (
  <div className="card">
    <div className="card_title" onClick=
      {this.toggleDetails.bind(this)}>
      {this.props.title}
    </div>
    {cardDetails}
  </div>
)
}
}

```

### 注意:

早期 React 版本(特别是使用 ES6 类之前的版本)内置了一个“魔法”功能,会自动将所有方法绑定到 this。由于这会给没有适应该功能的 JavaScript 开发人员带来困扰,因此被去除了。在目前的版本中,开发人员必须显式地将函数绑定到上下文。有几种方法可做到这一点,最简单的方法是直接使用 .bind(this) 来生成一个绑定函数。绑定和其他 JavaScript 实用方法会在附录 B 中进行讨论。

## 2.2 深入了解 JSX

JSX 是 React 为 JavaScript 语法带来的可选扩展,用于在 JavaScript 代码中编写声明式 XML 风格语法。

对于 Web 项目而言,React 的 JSX 提供了一组类似于 HTML 的 XML 标签,但在其他使用场景中,会使用其他组 XML 标签来描述用户界面(如 React with SVG、React Canvas 和 React Native)。转译后,XML 会被转换为针对 React 库的函数调用。这行代码:

```
<h1>Hello World</h1>
```

会被转译为:

```
React.createElement("h1", null, "Hello World");
```

JSX 是可选的。但拥抱它会带来如下好处:

- XML 包含特性的元素树非常适合表示 UI。
- 它能够更精确和更方便地呈现应用程序的结构。
- 它是普通 JavaScript,并不会改变这门语言的语义。

## 2.2.1 JSX 与 HTML

对于 Web 场景而言, JSX 看上去就像 HTML, 但它并不是 HTML 规范的具体实现。React 的创造者只是让 JSX 足够像 HTML, 这样就可以用来正确地描述 Web 界面, 并没有忽略这样一个事实, 即它仍然应该遵循 JavaScript 的风格和语法。

## 2.2.2 JSX 和 HTML 的不同之处

在使用 JSX 来编写 HTML 语法时, 你应当了解以下三个重要的方面:

- 标签特性采取驼峰式大小写风格。
- 所有元素都必须闭合。
- 特性名称基于 DOM API 而非 HTML 语言规范。

现在让我们来检查一下。

### 1. 标签特性采取驼峰式大小写风格

例如, 在 HTML 中, 输入标签可以包含一个可选的 `maxlength` 特性:

```
<input type="text" maxlength="30"/>
```

在 JSX 中, 该特性应该写作 `maxLength`(请注意大写字母“L”):

```
return <input type="text" maxLength="30" />
```

### 2. 所有元素都必须闭合

由于 JSX 是 XML, 因此元素都必须闭合。诸如 `<br>` 和 `<img>` 这样的标签并不包含结束标签, 需要自闭合。所以要使用 `<br/>` 而不是 `<br>`, 要使用 `<imgsrc="..." />` 而不是 `<imgsrc="...">`。

### 3. 特性名称基于 DOM API

这一点可能难以理解, 但实际上却非常简单。在与 DOM API 进行交互时, 标签特性的名称可能会和在 HTML 中使用时有所不同。其中一个例子是 `class` 和 `className`。例如, 对于这段普通的 HTML:

```
<div id="box" class="some-class"></div>
```

如果你想要使用普通 JavaScript 来操作 DOM 并更改它的类名, 你可能会编写这样的代码:

```
document.getElementById("box").className="some-other-class"
```

在 JavaScript 中, 这个特性称为 `className` 而不是 `class`。由于 JSX 只是 JavaScript 的一种语法扩展, 它遵循了 DOM 所定义的特性命名规范。同样的 `div` 用 JSX 来表示就

应该是：

```
return <div id="box" className="some-class"></div>
```

### 2.2.3 JSX 的怪异之处

JSX 偶尔也比较奇怪。针对在使用 JSX 构建组件时可能会遇到的常见问题，本节汇总了一些小技巧、提示和策略来供你应对。

#### 1. 单一根节点

React 组件只能渲染一个根节点。想要了解这个限制的原因，我们先来看看 `render` 函数的一个返回示例：

```
return (
  <h1>Hello World</h1>
)
```

它会被转换成一条语句：

```
return React.createElement("h1", null, "Hello World");
```

但是，下面的代码却不是合法的：

```
return (
  <h1>Hello World</h1>
  <h2>Have a nice day</h2>
)
```

需要明确的是，这并非 JSX 的限制，而是 JavaScript 的一个特性：一条返回语句只能返回单个值，而在前面的代码中我们尝试返回两条语句（两次 `React.createElement` 调用）。解决的方法非常简单：就像你在普通 JavaScript 中会做的那样，将所有返回值包含到一个根对象中。

```
return (
  <div>
    <h1>Hello World</h1>
    <h2>Have a nice day</h2>
  </div>
)
```

它完全有效，因为它会被转换成：

```
return React.createElement("div", null,
  React.createElement("h1", null, "Hello World"),
  React.createElement("h2", null, " Have a nice day"),
)
```

它返回单个值，并且是通过合法的 JavaScript 完成的。

## 2. 条件语句

如果语句不兼容于 JSX，看上去像是 JSX 的限制所致，实际上却是因为 JSX 只是普通的 JavaScript。为便于解释，让我们来回顾一下 JSX 是如何被转换为普通 JavaScript 的：如下所示的 JSX：

```
return (
  <div className="salutation">Hello JSX</div>
)
```

会被转换成这样的 JavaScript：

```
return (
  React.createElement("div", {className:"salutation"}, "Hello JSX");
)
```

然而，如果尝试在 JSX 的中间编写 if 语句，例如：

```
<div className={if (condition) { "salutation" }}>Hello JSX</div>
```

它就会被转换成一个非法的 JavaScript 表达式，如图 2-1 所示：

```
React.createElement("div", {className:if (condition) {"salutation"}}, "Hello JSX");
```

```
> React.createElement("div", {className: if (condition) { "salutation"}}, "Hello JSX");
✖ ▶ Uncaught SyntaxError: Unexpected token if
```

图 2-1 尝试在 JSX 中使用 if 表达式时产生的语法错误

## 3. 有什么解决方法？

尽管并无可能在 JSX 中使用“if”语句，但仍有根据条件渲染内容的方法，包括使用三元表达式和将条件赋值给一个变量(空值和未定义的值都会被 React 进行处理，JSX 在转义时什么都不会输出)。

### 使用三元表达式

如果你有一个非常简单的表达式，可以使用三元形式：

```
render() {
  return (
    <div className={condition ? "salutation" : ""}>
      Hello JSX
    </div>
  )
}
```

这段代码会被转换成一段合法的 JavaScript：

```
React.createElement("div", {className: condition ? "salutation" : ""}, "Hello JSX");
```

三元形式还可用来有条件地渲染整个节点：

```
<div>
  {condition ?
    <span>Hello JSX</span>
    : null}
</div>
```

### 将条件外置

如果三元表达式还不能应付你的要求，解决方法是不要在 JSX 的中间使用条件。简单地将条件语句移动到外部(就像你在第 2 章中隐藏和显示 `ContactItem` 细节时所采取的方法)。

下面是原先的代码：

```
render() {
  return (
    <div className={if (condition) { "salutation" }}>
      Hello JSX
    </div>
  )
}
```

将条件移动到 JSX 的外部，就像：

```
render() {
  let className;
  if(condition){
    className = "salutation";
  }
  return (
    <div className={className}>Hello JSX</div>
  )
}
```

React 知道如何处理未定义的值，如果条件为假，它甚至不会在 `div` 标签中创建 `class` 特性。

## 2.3 看板应用：指示卡片的打开和关闭状态

在第 1 章中，你使用了这种技术将条件外移并开关卡片明细。让我们使用三元形式来根据条件为卡片标题添加一个 `className`(为简洁起见，省略了一些原始代码)。最终结果如图 2-2 所示：

```
class Card extends Component {
  constructor() { ... }
  toggleDetails() { ... }
```



```

render() {
  let cardDetails;
  if (this.state.showDetails) {
    cardDetails = (
      <div className="card_details">
        {this.props.description}
        <CheckList cardId={this.props.id} tasks={this.props.tasks} />
      </div>
    );
  }

  return (
    <div className="card">
      <div className={
        this.state.showDetails? "card__title card__title--is-open" :
        "card__title"
      } onClick={this.toggleDetails.bind(this)}>
        {this.props.title}
      </div>
      {cardDetails}
    </div>
  )
}
}

```

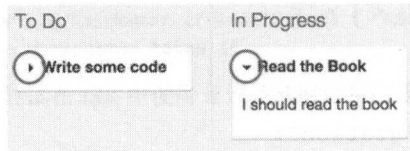


图 2-2 有条件的 class

### 2.3.1 空格

一个快速小提示: 在 HTML 中, 浏览器通常会在多行的元素之间渲染一个空格。React 的 JSX 则在你明确告诉它去渲染一个空格时, 才会这么做。例如, 下面的 JSX 会渲染成图 2-3 的样子。

```

return (
  <a href="http://google.com">Google</a> >
  <a href="http://facebook.com">Facebook</a>
)

```

GoogleFacebook

图 2-3 JSX 不会在行之间生成空格

想要显式地插入一个空格, 你可以使用包含一个空格字符串{" " }的表达式:

```

return(
  <a href="http://google.com">Google</a>{" "}

```

```
<a href="http://facebook.com">Facebook</a>
)
```

这样就会渲染出所期望的输出，如图 2-4 所示。

Google Facebook

图 2-4 使用表达式来渲染一个空格

### 2.3.2 JSX 中的注释

JSX 不是 HTML 这一事实产生的另一个怪异之处是它缺少对 HTML 注释的支持(例如 `<!--comment-->`)。尽管它不支持传统 HTML 标签的注释，但由于 JSX 由 JavaScript 表达式组成，它可以支持普通的 JavaScript 注释。需要注意，在处于一个标签的子区域中时，要用 `{}` 来包围注释。

```
let content = (
  <Nav>
    { /* child comment, put {} around */ }
    <Person
      /* multi
        line
        comment */
      name={window.isLoggedIn ? window.name : ''} // end of line comment
    />
  </Nav>
);
```

### 2.3.3 渲染动态 HTML

React 内置了 XSS 攻击保护措施，这意味着在默认情况下，它不允许动态生成 HTML 标签并附加到 JSX 中。通常而言这样做很好，但在某些特定情况下，你或许想在运行时生成 HTML。例如，在界面上渲染 Markdown 格式的数据。

注意：

Markdown 是一种格式，它允许你使用易读且易写的纯文本格式来写作。例如，将文本包含在两个星号内会将其加粗。

React 提供了 `dangerouslySetInnerHTML` 属性来跳过 XSS 保护并直接渲染任何内容。

### 2.3.4 看板应用：渲染 Markdown

让我们通过在看板应用的卡片描述中启用 Markdown 来实践这一操作。先从修改卡片描述开始，在其数据模型中添加一些 Markdown 格式。

```
let cardsList = [
```

```

{
  id:1,
  title:"Read the Book",
  description:"I should read the whole book",
  status:"in-progress",
  tasks: []
},
{
  id:2,
  title:"Write some code",
  description: "Code along with the samples in the book. The complete source
               can be found at [github](https://github.com/pro-react)",
  status:"todo",
  tasks: [
    {id:1,name:"ContactList Example",done:true},
    {id:2,name:"Kanban Example",done:false},
    {id:3,name:"My own experiments",done:false}
  ]
},
];

```

你需要一个 JavaScript 库将卡片描述中的 Markdown 转换为 HTML。有许多这样的开源库。在本例中，我们使用一个名为 `marked`(<https://github.com/chjj/marked>)的库。

如果你采取本书的示例并使用了模块系统，就可以在你的 `package.json` 中导入该库并安装它(可以通过一条命令 `npm install --save marked` 来完成这两步)。别忘了在你的文件开始处导入 `marked` 模块。

使用 `marked` 模块，你的代码看起来如下所示：

```

import React,{ Component }from 'react';
import CheckList from './CheckList';
import marked from 'marked';

```

然后，你就可以使用这个库所提供的 `marked()`函数将 Markdown 转换为 HTML(为简洁起见，我省略了一些与本例无关的代码)：

```

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}

  render() {
    let cardDetails;
    if (this.state.showDetails) {
      cardDetails = (
        <div className="card_details">
          {marked(this.props.description)}
          <CheckList cardId={this.props.id} tasks=
            {this.props.tasks}/>
        </div>

```

```
);
}

return (
  <div className="card">
    <div className={
      this.state.showDetails?"card_title card__
      title--is-open" : "card_title"
    } onClick={this.toggleDetails.bind(this)}>
      {this.props.title}
    </div>
    {cardDetails}
  </div>
)
}
```

然而正如预期，React 默认不允许在 JSX 内部渲染任何 HTML 标签，所以输出看上去如图 2-5 所示。

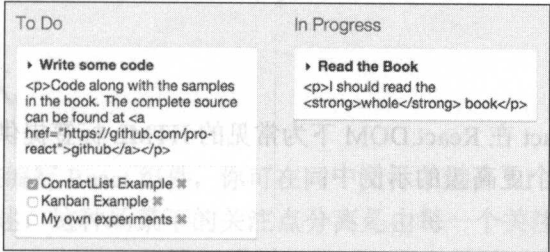


图 2-5 React 默认会转义 HTML

使用 dangerouslySetInnerHTML，你就可以实现期望的最终结果，如下面的代码和图 2-6 所示。

```
cardDetails = (
  <div className="card_details">
    <span dangerouslySetInnerHTML=
      {__html:marked(this.props.description)} />
    <CheckList cardId={this.props.id} tasks={this.props.tasks} />
  </div>
);
```

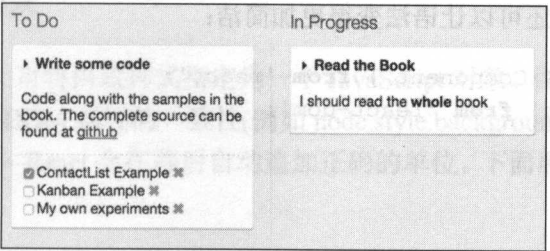


图 2-6 React 使用 dangerouslySetInnerHTML 来渲染动态生成的 HTML

## 2.4 脱离 JSX 的 React

JSX 使用一种精确而熟悉的语法来描述 UI 和树结构。它通过在不改变 JavaScript 语义的同时，又允许在 JavaScript 代码中使用 XML 做到了这一点。React 在设计时就考虑到了 JSX；但是，脱离 JSX 来使用 React 是绝对可能的。尽管在本书的其余示例中仍然会使用 JSX，但本节会简单介绍如何脱离 JSX 来使用 React。

### 2.4.1 普通 JavaScript 中的 React 元素

可使用 `react.createElement` 在普通 JavaScript 中创建 React 元素，它接受一个标签名称或组件、一个属性对象以及数量不定的可选子参数。

```
let child1 = React.createElement('li', null, 'First Text Content');
let child2 = React.createElement('li', null, 'Second Text Content');
let root = React.createElement('ul', {className: 'my-list'}, child1, child2);
React.render(root, document.getElementById('example'));
```

### 2.4.2 元素工厂

为方便起见，React 在 `React.DOM` 下为常见的 HTML 标签提供了工厂函数简写。让我们将它们应用到一个更高级的示例中：

```
React.DOM.form({className: "commentForm"},
  React.DOM.input({type: "text", placeholder: "Name"}),
  React.DOM.input({type: "text", placeholder: "Comment"}),
  React.DOM.input({type: "submit", value: "Post"})
)
```

上面的代码等效于下面的 JSX：

```
<form className="commentForm">
  <input type="text" placeholder="Name"/>
  <input type="text" placeholder="Comment"/>
  <input type="submit" value="Post"/>
</form>
```

使用解构赋值，还可以让语法变得更加简洁：

```
import React, { Component } from 'react';
import { render } from 'react-dom';

let {
  form,
  input
}=React.DOM;
```

```

class CommentForm extends Component {
  render() {
    return form({className:"commentForm"},
      input({type:"text",placeholder:"Name"}),
      input({type:"text",placeholder:"Comment"}),
      input({type:"submit",value:"Post"})
    )
  }
}

```

## 2.4.3 自定义工厂

还可为自定义组件创建工厂，例如：

```

let Factory =React.createFactory(ComponentClass);
...
let root =Factory({custom:'prop'});
render(root,document.getElementById('example'));

```

## 2.5 内联样式

通过使用 JSX 来编写 React 组件，你可在同一文件中组合 UI 定义(内容标记)和交互 (JavaScript)。如前所述，这种场景下的关注点分离是由每一个关注点所对应的组件带来的，这些组件各自分立、封装良好且可重用。但对于用户界面而言，除了内容和交互之外，还有一个重要因素：样式。

React 提供了使用 JavaScript 编写内联样式的能力。起初，使用 JavaScript 来编写样式的想法可能会有些奇怪，但相对于传统的 CSS，它又可以提供一些好处：

- 不需要选择器来限定样式的范围
- 避免特异性冲突
- 源顺序无关(Source Order Independence)

请注意 JavaScript 极具表现力，所以使用它你就可以自动获得变量、函数以及全方位的流程控制结构。

### 2.5.1 定义内联样式

在 React 组件中，可将内联样式指定为一个 JavaScript 对象。样式名称采取驼峰式大小写规则，以保持和 DOM 属性的一致性(例如 `node.style.backgroundImage`)。此外，并不需要指定像素单位——React 会在幕后自动追加正确的单位。下面展示了 React 中的内联样式示例：

```

import React, { Component } from 'react';
import {render} from 'react-dom';

```



```

class Hello extends Component {
  render() {
    let divStyle = {
      width: 100,
      height: 30,
      padding: 5,
      backgroundColor: '#ee9900'
    };

    return <div style={divStyle}>Hello World</div>
  }
}

```

## 2.5.2 看板应用：通过内联样式定义卡片颜色

使用 JavaScript 内联样式后，虽然也可以完全摆脱 CSS，但通常来说混合方式更合适。因为 CSS(或诸如 Sass 和 Less 这样的 CSS 预处理器)可以用于主要样式定义，而 React 组件中的内联样式则用于动态的、与状态相关的呈现。

在下面的步骤中，你会添加自定义颜色来标记一张卡片。

(1) 在数据模型中添加颜色。首先，让我们修改 CardList 数组来插入颜色：

```

let cardsList= [
  {
    id:1,
    title:"Read the Book",
    description:"I should read the book",
    color:'#BD8D31',
    status:"in-progress",
    tasks: []
  },
  {
    id:2,
    title:"Write some code",
    description:"Code along with the samples ... at [github]
      (https://github.com/pro-react)",
    color:'#3A7E28',
    status:"todo",
    tasks: [
      {id:1,name:"ContactList Example",done:true},
      {id:2,name:"Kanban Example",done:false},
      {id:3,name:"My own experiments",done:false}
    ]
  },
];

```

(2) 将颜色作为 props 传递给 Card 组件。Card 的父组件是一个 List 组件。目前，

List 组件会将三个特性作为 props 传递给 Card 组件：title、description 和 tasks。你需要将 color 作为一个新特性添加进去：

```
class List extends Component {
  render() {
    let cards = this.props.cards.map((card) => {
      return <Card id={card.id}
        title={card.title}
        description={card.description}
        color={card.color}
        tasks={card.tasks} />
    });

    return (
      ...
    )
  }
}
```

(3) 在 Card 组件中创建一个包含内联样式的 div。最后，你需要创建一个包含所有样式规则的对象，以及将要内联使用该样式对象的 div：

```
class Card extends Component {
  constructor() {...}
  toggleDetails() {...}

  render() {
    let cardDetails;
    if (this.state.showDetails) {...}

    let sideColor = {
      position: 'absolute',
      zIndex: -1,
      top: 0,
      bottom: 0,
      left: 0,
      width: 7,
      backgroundColor: this.props.color
    };

    return (
      <div className="card">
        <div style={sideColor}/>
        <div className={
          this.state.showDetails? "card__title card__title--is-open" :
            "card__title"
        } onClick={this.toggleDetails.bind(this)}>
          {this.props.title}
        </div>
      </div>
    )
  }
}
```

```
      {cardDetails}
    </div>
  )
}
}
```

图 2-7 展示了渲染结果。

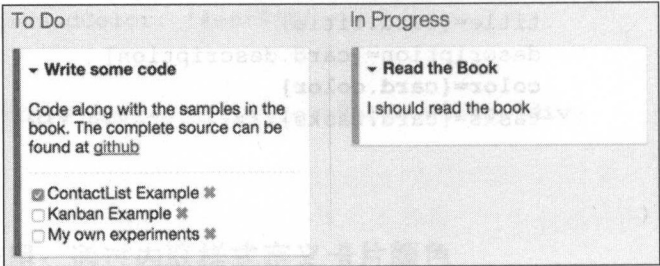


图 2-7 动态卡片颜色的内联样式

## 2.6 使用表单

在 React 中，组件的内部 state 要尽量精简，这是因为每次 state 发生变化时，组件都会重新进行渲染。这样做的目的是在 JavaScript 代码中准确地反映组件的 state，并让 React 保证 UI 同步。

为此，诸如

React 提供了两种方法来像处理组件那样处理表单，你可以根据你的应用特征或者个人喜好来进行选择。这两种处理表单域的方法采用受控组件的形式或非受控组件的形式。

### 2.6.1 受控组件

一个包含值或已选属性的表单组件称为受控组件。在一个受控组件中，元素内部所渲染的值将一直反映属性的值。默认情况下用户不能更改它。

看板卡片列表便属于这种情况。如果你尝试单击其中一个任务的复选框，它并不会发生变化。它们反映的是硬编码到 cardList 数组中的值，只有在你更改了数组本身后，它们才会发生变化。

在回到看板项目之前，让我们来看另一个示例。从包含一个输入域的 Search 组件开始：

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class Search extends Component {
  render() {
    return (
```

```
    <div>
      Search Term: <input type="search" value="React" />
    </div>
  )
}

render(<Search />, document.body);
```

以上代码渲染了一个表单域，用来显示一个不可变的值：“React”。在被渲染的元素上进行的任何用户输入都不会有效，因为 React 已经将该值声明为“React”，如图 2-8 所示。

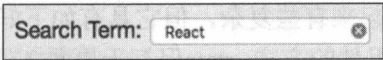


图 2-8 表单元素

要使该值发生变化，你需要将它作为一个组件 state 来处理。这样的话，该 state 值的任何变化就都能反映到界面上了。

```
class Search extends Component {
  constructor() {
    super();
    this.state = {
      searchTerm: "React"
    };
  }

  render() {
    return (
      <div>
        Search Term:
        <input type="search" value={this.state.searchTerm} />
      </div>
    )
  }
}
```

可使用 onChange 事件给予用户更新该 state 值的能力。

```
class Search extends Component {
  constructor() {
    super();
    this.state = {
      searchTerm: "React"
    };
  }

  handleChange(event) {
    this.setState({searchTerm: event.target.value});
  }
}
```

```
    }  
    render() {  
      return (  
        <div>  
          Search Term:  
          <input type="search" value={this.state.searchTerm}  
            onChange={this.handleChange.bind(this)} />  
        </div>  
      )  
    }  
  }  
}
```

这种操作表单的方法看起来有些复杂，但它具有如下好处：

- 它遵循了React处理组件的方式。state保存于界面之外，并且完全由你的JavaScript代码所管理。
- 这种模式让实现界面来响应或验证用户交互变得简单。例如，你可以非常轻松地限制用户最多只能输入 50 个字符：

```
this.setState({searchTerm:event.target.value.substr(0,50)});
```

2.6.2 特例

在创建受控表单组件时，有一些特例需要记住：textarea 和 select。

1. textarea

在 HTML 中，<textarea>的值通常通过其子元素进行设置：

```
<textarea>This is the description.</textarea>
```

对于 HTML 而言，这样做能让开发人员更便利地提供多行值。然而，由于 React 是 JavaScript，你并不会受到字符串的限制(例如，如果你想要换行，可使用\n)。为和其他表单元素保持一致，React 使用 value 属性来设置<textarea>的值：

```
<textarea value="This is a description." />
```

2. select

在 HTML 中，你使用选项标签上的 selected 特性来将选项设置为选中状态。在 React 中，为简化组件的操作，而采用了下面的格式：

```
<select value="B">  
  <option value="A">Mobile</option>  
  <option value="B">Work</option>  
  <option value="C">Home</option>  
</select>
```



### 2.6.3 非受控组件

非受控组件遵循 React 的原则，有自己的优势。尽管对于在 React 中构造的大多数其他组件而言，非受控组件是一种反模式，然而有时你并不需要逐个监管用户输入域。

较长的表单尤其如此，你只想让用户填写这些域，当用户完成后再处理所有内容。

非受控组件不为任何输入域提供值，渲染后的元素的值将反映用户的输入。例如：

```
return (
  <form>
    <div className="formGroup">
      Name: <input name="name" type="text" />
    </div>
    <div className="formGroup">
      E-mail: <input name="email" type="mail" />
    </div>
    <button type="submit">Submit</button>
  </form>
)
```

以上代码将渲染两个初始值为空的输入域。任何用户输入都会立即反映到渲染后的元素上。

提示：

如果你想要为非受控表单元素设置初始值，应使用 `defaultValue` 属性而非 `value` 属性。

可使用 `onSubmit` 来处理非受控组件表单，如下所示：

```
handleSubmit(event) {
  console.log("Submitted values are: ",
    event.target.name.value,
    event.target.email.value);
  event.preventDefault();
}

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <div className="formGroup">
        Name: <input name="name" type="text" />
      </div>
      <div className="formGroup">
        E-mail: <input name="email" type="mail" />
      </div>
      <button type="submit">Submit</button>
    </form>
  )
}
```

## 2.6.4 看板应用：创建一个任务表单

你的看板应用已经具备了一个受控组件：任务复选框。这次让我们来添加一个非受控组件：一个包含新任务的文本域。

```
class CheckList extends Component {
  render() {
    let tasks = this.props.tasks.map((task) => (...));
    return (
      <div className="checklist">
        <ul>{tasks}</ul>
        <input type="text"
          className="checklist--add-task"
          placeholder="Type then hit Enter to add a task" />
      </div>
    )
  }
}
```

由于没有指定 `value` 属性，用户可在该文本域内自由书写。在下一章中，你会将清单中的表单域连接起来，提供添加任务以及将任务标记为完成的功能。

让我们为该表单元素添加一些 CSS 样式来完成这一示例：

```
.checklist--add-task {
  border: 1px dashed #bbb;
  width: 100%;
  padding: 10px;
  margin-top: 5px;
  border-radius: 3px;
}
```

## 2.7 幕后的虚拟 DOM

正如你所看到的，到目前为止，React 的关键设计决策是让 API 看起来像是要在每次更新时都重新渲染整个应用程序。出于很多因素，DOM 操作是一个耗时的任务，所以为了提高性能，React 实现了一个虚拟 DOM。React 并没有在应用程序状态每一次发生变化时都更新真实的 DOM，而是创建了一个虚拟树，它看上去很像我想要的 DOM 状态。然后它会找出如何在不重新创建所有 DOM 节点的情况下，让 DOM 能够反映这一虚拟树。

为了让虚拟 DOM 树和真实 DOM 树保持一致，就需要找出必不可少的最小变化数量，这一过程称为子级校正(reconciliation)，而且通常来说，这是一个非常复杂且开销极大的操作。甚至在多次迭代和优化之后，它仍然是一个既困难又耗时的的问题。为处理这一问题，React 对典型应用程序的工作方式做了一些假设，允许在实际情况中使用一种速度更

快的算法。这些假设包括：

- 在比较 DOM 树中的节点时，如果节点类型不同(也就是说，将一个 div 变成一个 span)，React 就会将它们视为两个不同的子树，丢弃第一个，构建并插入第二个。
- 自定义组件也会使用相同的逻辑。如果它们类型不同，React 不会去尝试匹配它们的渲染内容，而将第一个从 DOM 中去除并插入第二个。
- 如果节点属于同一类型，React 可能采取两种方法来进行处理：
  - 如果是一个 DOM 元素(例如将 `<div id="before" />` 改为 `<div id="after" />`)，React 将只修改特性和样式(而不是替换元素树)。
  - 如果是一个自定义组件(例如将 `<Contact details={ false } />` 改为 `<Contact details={ true } />`)，React 不会去替换该组件，而将新属性发送给当前挂载的组件。最终会在组件上触发一次新的 `render()` 方法，这一过程还会使用新结果重新进行初始化。

## 2.7.1 key 属性

尽管 React 的虚拟 DOM 以及各种算法已经非常智能，在某些情况下为了更快捷，React 做一些假设并使用了比较朴素的方式。包含重复条目的列表尤其难以处理。为理解其原因，让我们来看看这个示例。代码清单 2-1 和代码清单 2-2 分别表示了列表的上一次渲染和当前渲染情况。

代码清单 2-1：示例列表

```
<li>Orange</li> <li>Banana</li>
```

代码清单 2-2：渲染后的示例列表

```
<li>Apple</li> <li>Orange</li>
```

这两个列表的差异非常明显，但将其中一个列表转换为另一个的最佳方案是什么？其中一个可行方案是在列表的最前面添加一个新条目(Apple)，然后删除最后一个条目(Banana)，但修改最后一个条目的名称和位置也是可行的。在较大列表中，会出现各种可行方案，每一种都可能导致副作用。考虑到节点可被插入、删除、替换和移动，就很难使用一种算法来确定所有可能性中最好的方案。

出于这样的原因，React 引入了 key 特性，key 是一个唯一标识符，允许你在需要进行插入、删除、替换和移动时快速查找元素。每次在循环中创建一个组件时，最好能为每个子元素都提供一个 key，这样就能帮助 React 库进行匹配而又能避免性能瓶颈了。

## 2.7.2 看板应用：key

你的上一个看板应用示例已经在浏览器的控制台中发出警告，指出子元素没有使用 key(参见图 2-9)。

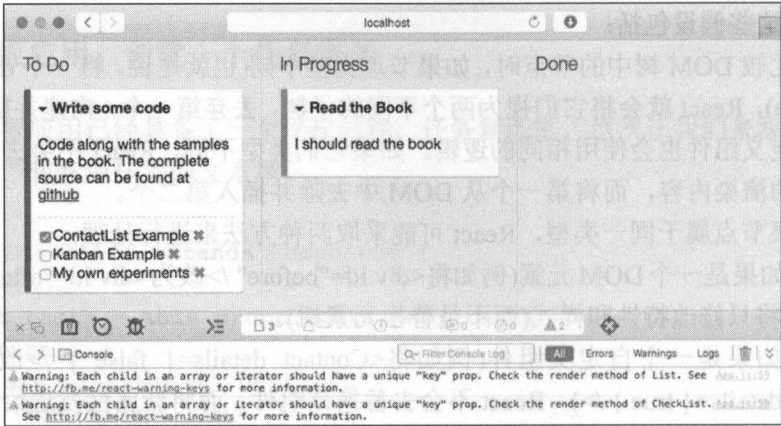


图 2-9 由于在 List 和 CheckList 组件中缺少 key 属性而发出的 React 警告

key 属性可以包含任何静态且唯一的值。在你的卡片数据中，每一张卡片都包含一个 ID。让我们在 List 组件中，将它用到 key 属性上：

```
class List extends Component {
  render() {
    let cards = this.props.cards.map((card) => {
      return <Card key={card.id}
        id={card.id}
        title={card.title}
        description={card.description}
        color={card.color}
        tasks={card.tasks} />
    });

    return (
      <div className="list">
        <h1>{this.props.title}</h1>
        {cards}
      </div>
    )
  }
}
```

在 CheckList 中还有一个数组。让我们也把 key 加进去：

```
class CheckList extends Component {
  render() {
    let tasks = this.props.tasks.map((task) => (
      <li key={task.id} className="checklist_task">
        <input type="checkbox" defaultChecked={task.done} />
        {task.name}{' '}
        <a href="#" className="checklist_task--remove" />
      </li>
    ));
  }
}
```

```

    return (...);
  }
}

```

### 2.7.3 refs

在 React 的运作方式中，渲染组件时，与你打交道的总是虚拟 DOM。例如，假设你更改了一个组件的 `state`，或将新的 `props` 发送给子元素，它们会被响应式地渲染到虚拟 DOM 中。然后 React 会在子级校正结束后更新真实 DOM。

这意味着作为开发人员，你永远不需要接触真实 DOM。虽然在某些情况下，你可能会发现自己想要接触组件渲染而成的真实 DOM 标记。但在操作真实 DOM 时，还是请三思而后行，因为在几乎所有情形中，你都可以在 React 模型中清晰明确地组织你的代码。在那些有必要操作真实 DOM(或者这样做能够带来一些好处)的少数情况下，React 提供了一个后门，称为 `refs`。

`refs` 可以像字符串属性那样用于任何组件，例如：

```
<input ref="myInput"/>
```

然后就可通过 `this.refs` 来访问被引用的 DOM 标记，就像：

```

let input =this.refs.myInput;
let inputValue =input.value;
let inputRect =input.getBoundingClientRect();

```

在本书中，我们很少会用到 `refs`，因为只有很少一些特定情况才会真正需要它。例如，让我们来创建一个组件，它只包含一个文本输入框和一个按钮，当按钮被单击后，让文本输入框获得焦点：

```

class FocusText extends Component {
  handleClick() {
    // Explicitly focus the text input using the raw DOM API.
    this.refs.myTextInput.focus();
  }
  render() {
    // The ref attribute adds a reference to the component to
    // this.refs when the component is mounted.
    return (
      <div>
        <input type="text" ref="myTextInput" />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.handleClick.bind(this)}
        />
      </div>
    );
  }
}

```



## 2.8 本章小结

在本章中，你学习了React的DOM抽象细节以及它实现高性能所采取的技术，例如事件委托及其差异(Diff)和子级校正特征(包括key属性的需求)。你还深入学习了JSX(以及如何根据需要脱离JSX使用React)、内联样式和表单。

# 使用组件构建应用程序

在第 2 章，我们对 React 进行了概述。如你所见，React 将一个基于组件的架构应用到 UI 界面构建的环节中。你已经了解到 React 的革新在于融合 HTML 和 JavaScript 来描述组件，并通过使用周全的、隔离的、可复用的、可组合的组件(而非使用不同的技术或语言)来实现概念的分离。

本章将讲述如何使用嵌套式组件来构建一个复杂的 UI 界面。你将看到通过 `propTypes` 来暴露一个组件 API 的重要性，并了解 React 应用程序中数据是如何流动的，并探索如何将组件组合到一起的技术。

## 3.1 校验属性

当创建组件时，记住它们可用于组合成更复杂的大组件，并且也可以被同项目、其他项目、其他开发人员复用。因为，显式地在你的组件中声明可以使用哪些属性、哪些属性是必需的、属性可以接受的数据类型等，是一个良好的实践。通过声明 `propTypes` 就可以做到这一点。`propTypes` 会帮助你“记录”你的组件，这样在未来的开发中，你将获得两个好处：

(1) 你可以很容易地打开一个组件的代码，查看哪些属性是必需的，它们的数据类型是什么。

(2) 当误用了组件时，React 会在控制台中显示一个错误信息，告诉你哪个属性有问题或是被误用了，以及是哪个 `render` 方法导致了问题。

`propTypes` 被定义成一个类构造函数属性。以 `Greeter` 组件为例：

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class Greeter extends Component {
  render() {
    return (
      <h1>{this.props.salutation}</h1>
    )
  }
}
```

```
render(<Greeter salutation="Hello World" />,
  document.getElementById('root'));
```

salutation 属性需要被设置为一个字符串，而且是必需的(如果没有为 salutation 属性提供任何值，就不能通过 render 使用它)。为做到这一点，你必须像前面那样，将 propTypes 定义为一个类构造函数属性：

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
```

```
class Greeter extends Component {
  render() {
    return (
      <h1>{this.props.salutation}</h1>
    )
  }
}
```

```
Greeter.propTypes = {
  salutation: PropTypes.string.isRequired
}
```

```
render(<Greeter salutation="Hello World" />,
  document.getElementById('root'));
```

如果在组件被实例化时，propTypes 的需求没有得到满足，会导致一次 console.warn 的调用。例如，如果你尝试不带任何属性来渲染一个 Greeter 组件：

```
React.render(<Greeter />, document.getElementById('root'));
```

就会出现下面的警告信息：

Warning:

Failed propType: Required prop `salutation` was not specified in `Greeter`.

对于可选属性，只要不设置.isRequired，React 就只有在这个属性被赋值的情况下才会检查它的数据类型。

### 3.1.1 属性的默认值

你还可为属性提供一个默认值，这样当属性未被显式赋值时，将使用这个默认值。设置默认值的语法和之前的类似：定义一个 defaultProps 对象作为一个构造函数属性。

例如，可将属性 salutation 设置为可选(将 isRequired 去掉即可)，然后将它的默认值设置为“Hello World”：

```
class Greeter extends Component {
  render() {
    return (
```

```
    <h1>{this.props.salutation}</h1>
  )
}

Greeter.propTypes = {
  salutation: PropTypes.string
}

Greeter.defaultProps = {
  salutation: "Hello World"
}

render(<Greeter />, document.getElementById('root'));
```

现在，如果没有为 `salutation` 属性提供任何值，你的组件就会输出一个默认的“Hello World”。如果提供了 `salutation` 属性的值，这个值就必须是字符串类型。

如前所述，你不一定需要在应用程序中使用 `propTypes`，但它提供了一种很好的方式来描述组件的 API 接口，总为组件指定 `propTypes` 是一个良好的实践。

### 3.1.2 内置的 propTypes 校验器

React 的 `propTypes` 包含有一组校验器(`validator`)，可用来确保组件接收到的数据是有效的。表 3-1 到 3-3 中包含的所有 `PropTypes` 是可选的，但是你可以将它们和 `isRequired` 组合在一起，来确保当没有提供属性时会显示一个警告信息。

表 3-1 JavaScript 基元 PropTypes 校验器

校验器	描述
PropTypes.array	属性必须是一个数组
PropTypes.bool	属性必须是一个布尔值(true/false)
PropTypes.func	属性必须是一个函数
PropTypes.number	属性必须是一个数字(或可以解析成数字的值)
PropTypes.object	属性必须是一个对象
PropTypes.string	属性必须是一个字符串

表 3-2 组合类型的 PropTypes 校验器

校验器	描述
PropTypes.oneOfType	属性必须属于指定的一组数据类型中的一种，例如： <pre>PropTypes.oneOfType([   PropTypes.string,   PropTypes.number,   PropTypes.instanceOf(Message) ])</pre>

(续表)

校验器	描述
PropTypes.arrayOf	属性必须是一种指定类型数据的数组，例如： <code>PropTypes.arrayOf(PropTypes.number)</code>
PropTypes.objectOf	属性必须是一个带有指定类型值的属性值的对象，例如： <code>PropTypes.objectOf(PropTypes.number)</code>
PropTypes.shape	属性必须是一个符合特定格式的对象。它需要拥有同一组属性，例如： <code>PropTypes.shape({   color: PropTypes.string,   fontSize: PropTypes.number })</code>

表 3-3 特别的 PropTypes 校验器

校验器	描述
PropTypes.node	属性必须是一个可以渲染的值：数字、字符串、元素或数组
PropTypes.element	属性必须是一个 React 元素
PropTypes.instanceOf	属性必须是指定类的实例(这里使用了 JavaScript 的 <code>instanceOf</code> 操作符)， 例如： <code>PropTypes.instanceOf(Message)</code>
PropTypes.oneOf	确保属性被限制为一组枚举值中的一项，例如： <code>PropTypes.oneOf(['News', 'Photos'])</code>

3.1.3 为看板应用定义 propTypes

为一个组件声明 propTypes 的正确方法，应该是在创建组件时就越早声明越好，但是由于直到现在我们才学习了 propTypes 并了解了它的重要性，所以现在检查一下看板应用的所有组件，并为它们声明 propTypes(如代码清单 3-1 到 3-4 所示)。

代码清单 3-1：为 KanbanBoard 组件声明 propTypes

```
import React, { Component, PropTypes } from 'react';
import List from './List';

class KanbanBoard extends Component {
  render() {...}
};
KanbanBoard.propTypes = {
```



```
cards: PropTypes.arrayOf(PropTypes.object)
};
```

```
export default KanbanBoard;
```

代码清单 3-2: 为 List 组件声明 propTypes

```
import React, { Component, PropTypes } from 'react';
import Card from './Card';
```

```
class List extends Component {
  render() {...}
};
List.propTypes = {
  title: PropTypes.string.isRequired,
  cards: PropTypes.arrayOf(PropTypes.object)
};
```

```
export default List;
```

代码清单 3-3: 为 Card 组件声明 propTypes

```
import React, { Component, PropTypes } from 'react';
import marked from 'marked';
import CheckList from './CheckList';
```

```
class Card extends Component {
  constructor() {...}
  toggleDetails() {...}
  render() {...}
};
Card.propTypes = {
  id: PropTypes.number,
  title: PropTypes.string,
  description: PropTypes.string,
  color: PropTypes.string,
  tasks: PropTypes.arrayOf(PropTypes.object)
};
```

```
export default Card;
```

代码清单 3-4: 位 Checklist 组件声明 propTypes

```
import React, { Component, PropTypes } from 'react';
```

```
class CheckList extends Component {
  render() {...}
};
```

```
CheckList.propTypes = {
  cardId: PropTypes.number,
```

```
tasks: PropTypes.arrayOf(PropTypes.object)
};
```

```
export default CheckList;
```

### 3.1.4 自定义 propTypes 校验器

我们已经了解到 React 提供了数量庞大的内置 propTypes 校验器, 这些校验器涵盖了每一个基本的场景, 但是有时仍会有某个场景需要一个更特别的校验器。

校验器本质上就是一个函数, 它接收的参数是一组属性、要检查的属性名称、组件的名称。函数要么什么都不返回(如果校验的属性值是有效的), 要么返回一个说明了无效属性值的 Error 对象。

**看板应用: 定义一个自定义的 propTypes 校验器**

在你的看板应用中, Card 组件有一个 title、一个 description 和其他属性。作为示例, 你将编写一个校验器, 当 title 属性值的长度超过 80 个字符时, 就显示一个警告信息。代码如代码清单 3-5 所示, 图 3-1 显示了没有通过此自定义 propTypes 校验器检查的例子。

代码清单 3-5: Card 组件的自定义 propTypes 校验器

```
import React, { Component, PropTypes } from 'react';
import marked from 'marked';
import CheckList from './CheckList';

let titlePropType = (props, propName, componentName) => {
  if (props[propName]) {
    let value = props[propName];
    if (typeof value !== 'string' || value.length > 80) {
      return new Error(
        `${propName} in ${componentName} is longer than 80 characters`
      );
    }
  }
}

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}
  render() {...}
}

Card.propTypes = {
  id: PropTypes.number,
  title: titlePropType,
  description: PropTypes.string,
  color: PropTypes.string,
```

```
tasks: PropTypes.arrayOf(PropTypes.object)
};

export default Card;
```

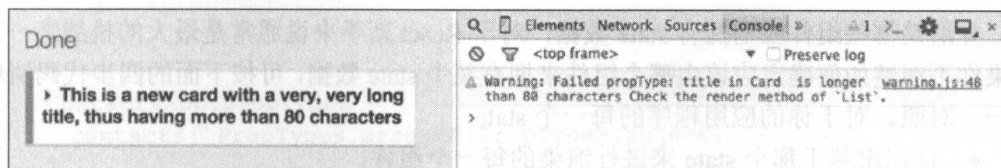


图 3-1 一个没有通过自定义校验器的检查的新卡片

#### 注意：

在上面的示例代码中，你使用了新的 JavaScript ES6 语法来组合字符串。你可以通过在线附录 C 更多地了解这个新特性，以及本书中用到的其他 ES6 语言特性的内容。

## 3.2 组件组合的策略与最佳实践

本节将涵盖通过将组件组合起来以创建 React 应用程序的策略和最佳实践。我们将讨论如何以结构化和有组织的方式，来实现状态的管理、数据的获取，以及对用户 UI 界面的控制。

### 3.2.1 有状态的组件和单纯组件

到目前为止，你已经了解到组件可以通过 `props` 和 `state` 来包含数据。

- `props` 是一个组件的配置信息。它们是从组件的上层传递而来，在组件接收到之后就保持不变。
- `state` 一开始在组件构造函数中被设置为一个默认值，然后随着组件的执行而不断变化(通常都是由于用户的操作而导致)。一个组件在内部管理它自己的 `state`，每当 `state` 变化时，组件就重新渲染。

对于 React 的组件而言，`state` 是可选的。在大部分 React 应用程序中组件分成两类：一类组件有 `state`(称为有状态的组件)，另一类则没有内部 `state`，只负责数据的显示(称为单纯组件)。

单纯组件只接收 `props`，然后负责将这些 `props` 渲染到一个视图中。这个特性使得单纯组件很容易被复用和被测试。

然而，有时你需要响应用户的输入，或是一个服务器请求，或是时间的流逝。这样你就需要使用 `state`。有状态的组件通常具有更高的组件层级，也就是说，它们通常包装了一个或多个有状态或单纯组件。

让一个应用程序的大部分组件都是无状态的单纯组件是一个最佳实践。如果应用程序的 `state` 在多个组件之间传播，会导致它难以跟踪。由于应用程序的工作方式变得更不清晰，也降低了程序代码的可预测性。这会在你的代码中潜在地引入一些难以理清和解

决的情况。

### 3.2.2 哪些组件应当是有状态组件

弄清楚哪些组件应当拥有 `state` 数据, 对于 React 新手来说通常是最大的挑战之一。如果你不清楚如何确定应该由哪个组件来拥有某个 `state` 数据, 可按下面的四步代码清单来一一对照。对于你的应用程序的每一个 `state`:

- 标识出基于那个 `state` 来进行渲染的每一个组件。
- 找到一个通用的 `owner` 组件(在组件层级中, 位于所有需要那个 `state` 的组件之上的单个组件)。
- 通用的 `owner` 组件或者更高层级的另一个组件应当拥有那个 `state`。
- 如果你无法找到一个合理的拥有 `state` 的组件, 创建一个新组件来拥有 `state`, 然后在组件层级中, 将它添加到所有需要那个 `state` 的组件上的某个位置。

为演示上面的步骤, 下面创建一个非常简单的联系人应用程序, 如图 3-2 所示。

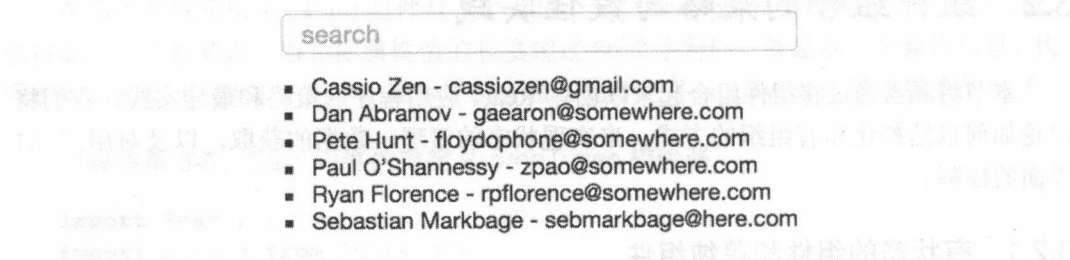


图 3-2 带搜索功能的示例联系人应用程序

这个应用程序的组件层级是:

- **ContactsApp**: 主组件
  - **SearchBar**: 显示一个用户可用来筛选联系人的输入框
  - **ContactsList**: 遍历数据, 创建一系列 **ContactItem** 组件
  - **ContactItem**: 显示联系人数据

在代码中, 联系人列表数据保存在一个全局变量中。在实际应用程序中, 数据通常从远端获取, 当前为了简化这个示例, 我们就硬编码数据好了。代码清单 3-6 显示了包含 **ContactsApp**、**SearchBar**、**ContactList**、**ContactItem** 组件(和它们的 `propTypes`)的完整代码。

#### 代码清单 3-6: 联系人应用程序的代码

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';

// Main component. Renders a SearchBar and a ContactList
class ContactsApp extends Component {
  render() {
    return(
```

```

    <div>
      <SearchBar />
      <ContactList contacts={this.props.contacts} />
    </div>
  )
}

ContactsApp.propTypes = {
  contacts: PropTypes.arrayOf(PropTypes.object)
}

class SearchBar extends Component {
  render() {
    return <input type="search" placeholder="search" />
  }
}

class ContactList extends Component {
  render() {
    return (
      <ul>
        {this.props.contacts.map(
          (contact) => <ContactItem key={contact.email}
                                name={contact.name}
                                email={contact.email} />
        )}
      </ul>
    )
  }
}

ContactList.propTypes = {
  contacts: PropTypes.arrayOf(PropTypes.object)
}

class ContactItem extends Component {
  render() {
    return <li>{this.props.name} - {this.props.email}</li>
  }
}

ContactItem.propTypes = {
  name: PropTypes.string.isRequired,
  email: PropTypes.string.isRequired,
}

let contacts = [
  { name: "Cassio Zen", email: "cassiozen@gmail.com" },
  { name: "Dan Abramov", email: "gaearon@somewhere.com" },
  { name: "Pete Hunt", email: "floydophone@somewhere.com" },
  { name: "Paul O'Shannessy", email: "zpao@somewhere.com" },
]

```



```

    { name: "Ryan Florence", email: "rpflorence@somewhere.com" },
    { name: "Sebastian Markbage", email: "sebmarkbage@here.com" },
  ]

  render(<ContactsApp contacts={contacts} />,
    document.getElementById('root')));

```

到现在为止，上面这个应用程序中的所有组件都是单纯组件；它们只渲染通过 props 接收到的数据。然而，你需要为你的应用程序添加一个筛选功能，你将需要存储可变的数据来实现。下面通过上面那个四步代码清单，找出应当把 state 放到应用程序的哪个地方。

ContactList 组件需要基于 state 来筛选联系人，SearchBar 组件需要显示出搜索文本。通用的 owner 组件是 ContactsApp。

将筛选文本存储为 ContactsApp 组件的 state 从概念上是符合逻辑的。ContactsApp 之后会将筛选文本以 props 的方式向下传递给子组件。SearchBar 组件将使用其作为文本框的值，ContactList 组件将使用它筛选联系人数据。下面来逐个改造组件(如代码清单 3-7 到 3-9 所示)。在代码清单 3-8 中，SearchBar 组件将通过 props 接收到 filterText，然后将这个 props 设置为文本框的值。文本框现在是一个受控表单组件(如第 2 章所述)。在代码清单 3-9 中，ContactList 组件同样通过 props 接收到 filterText，然后基于它的值来筛选要显示的联系人。

#### 代码清单 3-7：更新后的有状态 ContactsApp 组件

```

class ContactsApp extends Component {
  constructor() {
    super();
    this.state = {
      filterText: ''
    };
  }
  render() {
    return (
      <div>
        <SearchBar filterText={this.state.filterText} />
        <ContactList contacts={this.props.contacts}
          filterText={this.state.filterText}/>
      </div>
    )
  }
}
ContactsApp.propTypes = {...}

```

#### 代码清单 3-8：The SearchBar Component

```

class SearchBar extends Component {
  render() {
    return <input type="search" placeholder="search"
      value={this.props.filterText} />

```

```

    }
  }
  // Don't forget to add the new propTypes requirements
  SearchBar.propTypes = {
    filterText: PropTypes.string.isRequired
  }
}

```

### 代码清单 3-9: ContactList 组件

```

class ContactList extends Component {
  render() {
    let filteredContacts = this.props.contacts.filter(
      (contact) => contact.name.indexOf(this.props.filterText) !== -1
    );
    return (
      <ul>
        {filteredContacts.map(
          (contact) => <ContactItem key={contact.email}
                                name={contact.name}
                                email={contact.email} />
        )}
      </ul>
    )
  }
}

```

现在你的应用程序就只有在层级顶部有一个有状态组件，其他三个组件则都是只通过 `props` 接受要显示的数据的单纯组件。`ContactList` 组件基于 `filterText` 属性来筛选要显示的数据(你现在可以尝试修改代码中 `ContactsApp` 的 `filterText` 的值)，但是用户无法在搜索框中输入任何内容，因为在 `SearchBar` 组件中无法修改它的 `state`；`state` 被父组件所拥有。

在下一节，你将学习在(单纯)子组件中如何与(有状态的)父组件进行通信。

### 3.2.3 数据流和组件通信

在一个 React 应用程序中，数据沿着组件的层级，从上层传递到下层：React 让这个数据流动的过程显得很直观，以使得开发人员很容易理解程序是如何工作的。

然而，在有意义的应用程序中，嵌套的子组件需要能与父组件通信。实现这种通信的一个方法，就是通过由父组件作为 `props` 传递而来的回调进行。

下面使用 `ContactsApp` 示例来进行演示。`state` 属于顶层的 `ContactsApp` 组件，它通过 `props` 向下传递给下层的 `SearchBar` 和 `ContactList` 组件。

你想要确保当用户无论何时更改搜索表单时，都能基于用户输入的更新的搜索文本来更新 `state`。既然组件只应该更新它们自己的 `state`，`ContactsApp` 将传递一个回调给 `SearchBar`，这个回调会在需要更新 `state` 时被调用。你可在 `SearchBar` 组件中使用输入框的 `onChange` 事件，以在输入框内容被修改时得到通知。在 `ContactsApp` 组件中，你创建

一个本地函数，来修改 `filterText` 状态，并将这个函数作为一个 `prop` 传递给 `SearchBox` 组件(如代码清单 3-10 所示)。

代码清单 3-10: 创建一个本地函数

```
class ContactsApp extends Component {
  constructor() {...}

  handleUserInput(searchTerm) {
    this.setState({filterText: searchTerm})
  }

  render() {
    return (
      <div>
        <SearchBar filterText={this.state.filterText}
          onUserInput={this.handleUserInput.bind(this)} />
        <ContactList contacts={this.props.contacts}
          filterText={this.state.filterText}/>
      </div>
    )
  }
}

ContactsApp.propTypes = {
  contacts: PropTypes.arrayOf(PropTypes.object)
}
```

`SearchBox` 组件将回调函数作为一个 `prop` 接收，并在输入框的 `onChange` 事件中调用这个接收到的回调(如代码清单 3-11 所示)。

代码清单 3-11: 接收回调并在 `onChange` 事件中调用它

```
class SearchBar extends Component {
  handleChange(event) {
    this.props.onUserInput(event.target.value)
  }

  render() {
    return <input type="search"
      placeholder="search"
      value={this.props.filterText}
      onChange={this.handleChange.bind(this)} />
  }
}

SearchBar.propTypes = {
  onUserInput: PropTypes.func.isRequired,
  filterText: PropTypes.string.isRequired
}
```

即时搜索(search in action)的效果和完整的代码如图 3-3 和代码清单 3-12 所示。

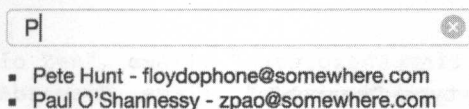


图 3-3 联系人应用程序的即时搜索功能

### 代码清单 3-12: 联系人应用程序的完整代码

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';

// Main (stateful) component.
// Renders a SearchBar and a ContactList
// Passes down filterText state and handleUserInput callback as props
class ContactsApp extends Component {
  constructor() {
    super();
    this.state = {
      filterText: ''
    };
  }

  handleUserInput(searchTerm) {
    this.setState({filterText: searchTerm})
  }

  render() {
    return (
      <div>
        <SearchBar filterText={this.state.filterText}
          onUserInput={this.handleUserInput.bind(this)} />
        <ContactList contacts={this.props.contacts}
          filterText={this.state.filterText}/>
      </div>
    )
  }
}

ContactsApp.propTypes = {
  contacts: PropTypes.arrayOf(PropTypes.object)
}

// Pure component that receives 2 props from the parent
// filterText (string) and onUserInput (callback function)
class SearchBar extends Component {
  handleChange(event) {
```

```

    this.props.onUserInput(event.target.value)
  }

  render() {
    return <input type="search"
      placeholder="search"
      value={this.props.filterText}
      onChange={this.handleChange.bind(this)} />
  }
}

SearchBar.propTypes = {
  onUserInput: PropTypes.func.isRequired,
  filterText: PropTypes.string.isRequired
}

// Pure component that receives both contacts and filterText as props
// The component is responsible for actually filtering the
// contacts before displaying them.
// It's considered a pure component because given the same
// contacts and filterText props the output will always be the same.
class ContactList extends Component {
  render() {
    let filteredContacts = this.props.contacts.filter(
      (contact) => contact.name.indexOf(this.props.filterText) !== -1
    );
    return (
      <ul>
        {filteredContacts.map(
          (contact) => <ContactItem key={contact.email}
            name={contact.name}
            email={contact.email} />
        )}
      </ul>
    )
  }
}

ContactList.propTypes = {
  contacts: PropTypes.arrayOf(PropTypes.object),
  filterText: PropTypes.string.isRequired
}

class ContactItem extends Component {
  render() {
    return <li>{this.props.name} - {this.props.email}</li>
  }
}

ContactItem.propTypes = {
  name: PropTypes.string.isRequired,
  email: PropTypes.string.isRequired
}

```



```
}

let contacts = [
  { name: "Cassio Zen", email: "cassiozen@gmail.com" },
  { name: "Dan Abramov", email: "gaearon@somewhere.com" },
  { name: "Pete Hunt", email: "floydophone@somewhere.com" },
  { name: "Paul O'Shannessy", email: "zpao@somewhere.com" },
  { name: "Ryan Florence", email: "rpflorence@somewhere.com" },
  { name: "Sebastian Markbage", email: "sebmarkbage@here.com" },
]

render(<ContactsApp contacts={contacts} />,
  document.getElementById('root'));
```

### 3.3 组件的生命周期

在创建 React 组件时，可声明一些函数，并让它们在组件的生命周期中的某些特定时间点被调用。了解每一个组件生命周期函数所扮演的角色以及它们的顺序，将让你可以在一个组件被创建或销毁时执行特定的操作。同时你还可根据需要去响应 props 或 state 的变化。

此外，了解有关组件生命周期函数的知识，对于进行性能优化(将在第 7 章介绍)和在 Flux 架构中组织你的组件(将在第 6 章介绍)也是必需的。

#### 3.3.1 声明周期的阶段与函数

要对生命周期有一个清晰认识，你需要区分下面这几个阶段：初始组件创建阶段、state 和 props 更改、触发更新、组件的卸载阶段。图 3-4 到图 3-7 演示了在每个阶段会被调用的函数。

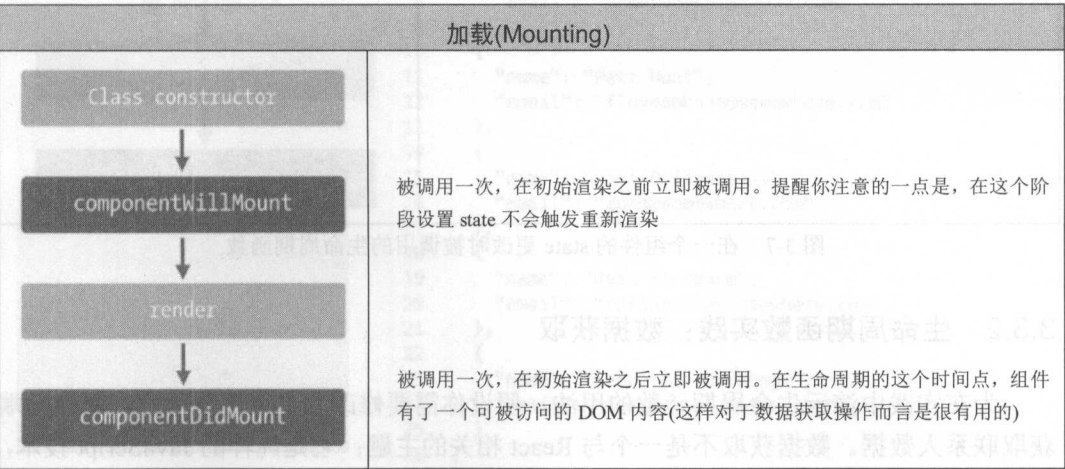


图 3-4 在挂载阶段中被调用的生命周期函数

卸载(Unmounting)	
<code>componentWillUnmount</code>	当一个组件被从DOM中卸载时被立即调用。当你需要进行一些清理操作，例如移除在加载阶段定义的事件侦听计时器，就可以利用此函数

图 3-5 在卸载阶段中被调用的生命周期函数

props 更改	
<div><div>componentWillReceiveProps</div><div>↓</div><div>shouldComponentUpdate</div><div>↓</div><div>componentWillUpdate</div><div>↓</div><div>render</div><div>↓</div><div>componentDidUpdate</div></div>	<p>当一个组件接收到新的props时被调用。在这个函数中调用this.setState()不会触发一次额外的渲染</p> <p>shouldComponentUpdate是一个特别的函数，它会在render函数之前被调用，在这个函数里面可以定义是否需要进行一次渲染，或者还是说这次渲染可以被跳过。这个函数对于性能优化而言很有用，有关细节将在第9章中讲述</p> <p>当接收到新的props或者state而进行渲染之前，此函数被立即调用。在这个函数中不允许通过this.setState对state进行任何更改，它只能严格地用于准备即将开始的渲染，而不能自己再去触发一次更新</p> <p>在组件的更新被刷新到DOM之后，就被立即调用。</p>

图 3-6 在一个组件的 props 更改时被调用的生命周期函数

state 更改	
<p>state更改基本上会触发和props更改相同的那些生命周期函数，只有一个例外：没有对应于componentWillReceiveProps的函数。一个传入props的转换操作可能会导致state的更改，但反之则不然。如果你需要在state更改时执行操作，使用componentWillUpdate函数</p>	<div><div>shouldComponentUpdate</div><div>↓</div><div>componentWillUpdate</div><div>↓</div><div>render</div><div>↓</div><div>componentDidUpdate</div></div>

图 3-7 在一个组件的 state 更改时被调用的生命周期函数

3.3.2 生命周期函数实践：数据获取

为在实践中演示生命周期函数的用法，假设你需要修改联系人应用程序，来从远端获取联系人数据。数据获取不是一个与 React 相关的主题；它是纯粹的 JavaScript 技术，但要注意，你必须在一个特定的组件生命周期里面获取数据：componentDidMount 生命周期函数。

既然本章讲述的是有关组件组合的策略和最佳实践，所以也应当提醒你一下，应当避免将数据获取的逻辑添加到已经负责其他事项的组件上。相反，作为一项最佳实践，应当创建一个新的有状态组件，来专门负责和远端 API 通信，并将数据和回调以 `props` 的方式传递给下层组件。有些人将这种类型的组件称为“容器组件”。

你将在你的联系人应用程序中，使用容器组件这个概念，而将数据逻辑添加到现有的 `ContactsApp` 组件上。你将在顶层创建一个名为 `ContactsAppContainer` 的新组件。现有的 `ContactsApp` 将不做改变。它将继续使用 `props` 来接收数据。

#### 注意：

在这个示例代码中，你将使用新的 `window.fetch` 函数，这个函数比使用 `XMLHttpRequest` 更易于进行 Web 请求和处理返回的响应数据。在撰写本书时，只有 Chrome 和 Firefox 支持这个新的标准，所以你需要从 npm 安装并导入 `whatwg-fetch` 这个“替代品” (Polyfill) (Polyfill 是用在浏览器中的一个术语，表示当浏览器不内置支持某个功能时，可以通过引入某个其他替代品，以使得浏览器能支持这个功能)。

```
npm install --save whatwg-fetch
```

下面开始将硬编码的数据移动到一个 json 文件中(这个 json 文件必须放到 `public` 或 `static` 文件夹中，这样才能被开发服务器所支持)，如图 3-8 所示。你的项目文件夹结构可能和图片上所显示的不同；需要着重提醒的是，位于 `public` 或 `static` 文件夹里面的文件将由 Web 服务器来支持。

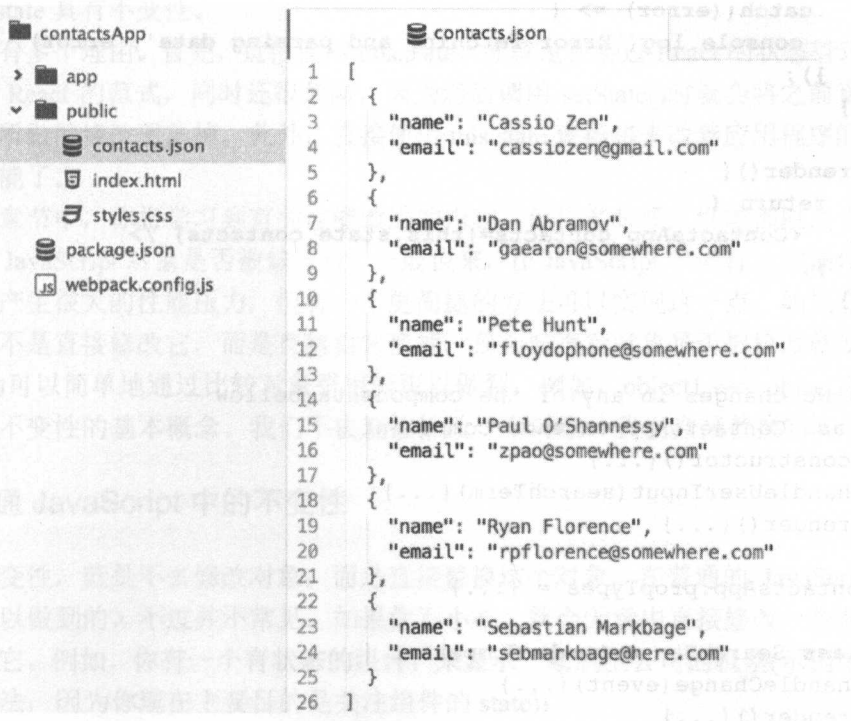


图 3-8 新的 `contacts.json` 文件

新的 `ContactsAppContainer` 组件如代码清单 3-13 所示。没有其他组件需要被修改，只不过之前是要渲染 `ContactsApp`，现在则是渲染 `ContactsAppContainer` 到文档中(如代码清单的最后几行所示)。

代码清单 3-13: 新的 `ContactsAppContainer` 组件

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import 'whatwg-fetch';

class ContactsAppContainer extends Component {
  constructor() {
    super();
    this.state = {
      contacts: []
    };
  }

  componentDidMount() {
    fetch('./contacts.json')
      .then((response) => response.json())
      .then((responseData) => {
        this.setState({contacts: responseData});
      })
      .catch((error) => {
        console.log('Error fetching and parsing data', error);
      });
  }

  render() {
    return (
      <ContactsApp contacts={this.state.contacts} />
    );
  }
}

// No changes in any of the components bellow
class ContactsApp extends Component {
  constructor() {...}
  handleUserInput(searchTerm) {...}
  render() {...}
}
ContactsApp.propTypes = {...}

class SearchBar extends Component {
  handleChange(event) {...}
  render() {...}
}
SearchBar.propTypes = {...}
```

```

class ContactList extends Component {
  render() {...}
}
ContactList.propTypes = {...}

class ContactItem extends Component {
  render() {...}
}
ContactItem.propTypes = {...}

// You now render ContactAppContainer, instead of ContactsApp
render(<ContactAppContainer />, document.getElementById('root'));

```

这就是要实现远程数据获取所需要做的所有工作了。如果你在浏览器中重新加载联系人应用程序，它看起来和之前没有差别，但在底层，现在它是从一个外部数据源中加载联系人数据了。

## 3.4 浅谈不变性

React 提供了一个 `setState` 方法来修改组件内部的状态。要注意，应当总是使用 `setState` 方法来更新组件 UI 的状态，而不应直接修改 `this.state`。作为一项经验之谈，应当认为 `this.state` 具有不变性。

这么做有多个理由。首先，直接使用 `this.state`，你就是在绕过 React 的状态管理，这不仅违反了 React 的范式，同时还很危险，因为随后调用 `setState()` 时就会将之前直接通过 `this.state` 所做的修改覆盖掉。此外，直接使用 `this.state` 使得将来改善应用程序的性能变得不太可能了。

在后续章节中，你将学习到有关性能改进的内容，但很多时候，性能都取决于比较和检查一个 JavaScript 对象是否被修改过。一般说来，在 JavaScript 中进行这种操作都代价高昂，会产生很大的性能压力，但有一个更简捷的方法可以实现这一点：如果在对象被修改时，不是直接修改它，而是直接将它换掉，那么要查看对象是否被修改就变得快很多了(因为可以简单地通过比较对象引用就可以做到，例如，`object1 === object2`)。

这就是不变性的基本概念。我们不去直接修改一个对象，我们直接替换它。

### 3.4.1 普通 JavaScript 中的不变性

所谓不变性，就是不去修改对象，而是直接替换这个对象，在普通的 JavaScript 中这是绝对可以做到的，不过并不常见。如果你不小心，就会无意中直接修改一个对象，而没有替换它。例如，你有一个有状态的组件，来显示一家航空公司的机票(示例中省略了 `render` 方法，因为你现在主要目的是关注组件的 `state`)：

```
import React, { Component } from 'react';
```



```

import { render } from 'react-dom';

class Voucher extends Component {
  constructor() {
    super(...arguments)
    this.state = {
      passengers:[
        'Simmon, Robert A.',
        'Taylor, Kathleen R.'
      ],
      ticket:{
        company: 'Dalta',
        flightNo: '0990',
        departure: {
          airport: 'LAS',
          time: '2016-08-21T10:00:00.000Z'
        },
        arrival: {
          airport: 'MIA',
          time: '2016-08-21T14:41:10.000Z'
        },
        codeshare: [
          {company:'GL', flightNo:'9840'},
          {company:'TM', flightNo:'5010'}
        ]
      }
    }
  }

  render() {...}
}

```

现在，假设你想要将一个新乘客添加到 `passengers` 数组中。如果不小心，就会无意中直接修改组件的 `state` 对象。例如：

```

let updatedPassengers = this.state.passengers;
updatedPassengers.push('Mitchell, Vincent M.');
```

```

this.setState({passengers:updatedPassengers});
```

上面这段代码的问题，你可能也已经猜到了，就是在JavaScript中，对象和数组都是以引用方式传递的。这意味着当执行 `updatedPassengers=this.state.passengers` 时，你并没有创建数组的一个副本；你只不过创建了一个新引用，而引用指向的是当前组件的 `state` 中的那个数组。因此，在对数据引用调用 `push` 方法时，你实际上就是在直接修改 `state`。

要在 JavaScript 中创建一个数组的实际副本，你需要使用非侵入式的方法，也就是说，使用那些会返回一个新数组，而非直接对原数组进行修改的方法。`map`、`filter`、`concat` 都是这种形式的非侵入式数组方法。下面再来实现一次向数组中添加一个新乘客的操作，这次，我们使用 `Array` 的 `concat` 方法：

```
// updatedPassengers is a new array, returned from concat
let updatedPassengers = this.state.passengers.concat('Mitchell, Vincent M. ');
this.setState({passengers:updatedPassengers});
```

在 JavaScript 中基于原来的对象生成新的对象还有其他一些方式，例如使用 `Object.assign`。`Object.assign` 会将所有给定对象上的所有属性合并到目标对象：

```
Object.assign(target, source_1, ..., source_n)
```

它首先遍历 `source_1` 对象的所有属性，将它们复制到 `target` 对象上，然后对 `source_2` 和之后的所有对象做同样的事情。例如，要修改 `state.ticket` 的 `flightNo`，你可以这样做：

```
// updatedTicket is a new object with the original properties of
// this.state.ticket
// merged with the new flightNo.
let updatedTicket = Object.assign({}, this.state.ticket,
  {flightNo:'1010'});
this.setState({ticket:updatedTicket});
```

注意：

在撰写本书时，只有 Chrome 和 Firefox 支持新的 `Object.assign` 方法，但是好消息是 Babel(你正在和 Webpack 一块儿使用的 ES6 编译器)已经为其他浏览器提供了替代品。你需要做的就是使用 “`npm install --save babel-polyfill`” 进行安装，并使用 “`import 'babel-polyfill'`” 将其导入。

### 3.4.2 嵌套对象

虽然在大多数场景中，数组的非侵入式方法和 `Object.assign` 已经可以完成我们想要的操作，但是如果你的 `state` 中包含嵌套对象或数组，那么情况就会变得更棘手。导致棘手的原因是因为 JavaScript 语言的一个特性：对象和数组都通过引用方式传递，并且数组的非侵入式方法和 `Object.assign` 都不会做深度复制(deep copies)。在实践中，这意味着在你通过数组非侵入式方法和 `Object.assign` 所创建的新对象中所包含的嵌套对象和数组，仍然指向的是原对象中的嵌套对象和数组。

下面基于实际的代码来看看会发生什么，仍然用之前的那个机票对象做例子：

```
let originalTicket={
  company: 'Delta',
  flightNo: '0990',
  departure: {
    airport: 'LAS',
    time: '2016-08-21T10:00:00.000Z'
  },
  arrival: {
    airport: 'MIA',
    time: '2016-08-21T14:41:10.000Z'
  },
},
```

```
codeshare: [
  {company: 'GL', flightNo: '9840'},
  {company: 'TM', flightNo: '5010'}
]
```

如果使用 `Object.assign` 创建一个机票对象，如下所示：

```
let newTicket = Object.assign({}, originalTicket, {flightNo: '5690'})
```

在内存中就会存在两个对象，如图 3-9 所示。

```
originalTicket
▶ Object {company: "Delta", flightNo: "0990", departure: Object, arrival: Object}
newTicket
▶ Object {company: "Delta", flightNo: "5690", departure: Object, arrival: Object}
```

图 3-9 可以看到，`originalTicket` 和 `newTicket` 对象的 `flightNo` 属性的值是不同的

然而，由于默认 JavaScript 是以引用方式来传递数组和对象，所以 `newTicket` 对象中的 `departure` 和 `arrival` 对象并不是新的副本；它们都指向 `originalTicket` 对象中的相同嵌套对象。如果你尝试修改 `newTicket` 上的 `arrival` 对象，如下所示：

```
newTicket.arrival.airport="MCO"
```

图 3-10 显示了 `originalTicket` 和 `newTicket` 对象的当前状态。

```
originalTicket
▼ Object {company: "Delta", flightNo: "0990", departure: Object, arrival: Object}
  ▼ arrival: Object
    airport: "MCO"
    time: "2016-08-21T14:41:10.000Z"
newTicket
▼ Object {company: "Delta", flightNo: "5690", departure: Object, arrival: Object}
  ▼ arrival: Object
    airport: "MCO"
    time: "2016-08-21T14:41:10.000Z"
```

图 3-10 `originalTicket` 和 `newTicket` 对象的 `arrival` 属性都引用了同一个对象

同样，这些行为都和 React 无关；这只不过是 JavaScript 的默认行为，但是这种默认行为在你想要修改一个带有嵌套对象的 React 组件 `state` 时，就会影响到 React。你可以尝试基于原有对象创建一个深度副本，但是这不是一个好的选择，因为深度副本会产生较大的性能损失，在有些场景中，进行深度副本甚至都是不可能的。好消息是对于这个问题有一个简单的解决方案：React 插件包提供了一个辅助函数(称为不变性助手)来帮助你更新更复杂的、嵌套的模型对象。

### 3.4.3 React 不变性助手

React 的插件包提供了一个不变性助手：`update`。这个 `update` 函数应用到普通的

JavaScript对象和数组之上，帮助你将它们包装成不可变的对象：函数不会真正修改这些对象，而总是返回一个新的可变的对象。

要使用这个函数，需要安装和依赖这个库：

```
npm install -save react-addons-update
```

然后，在 JavaScript 文件中，像这样来导入它：

```
import update from 'react-addons-update';
```

update 函数接受两个参数。第一个参数是你想要更新的对象或数组。第二个参数是一个对象，它描述了你想要在何处进行何种修改。用这个简单的对象来作为例子：

```
let student = {name: 'John Caster', grades: ['A', 'C', 'B']}
```

要创建这个对象的一个修改 grades 后的副本，update 函数的用法如下：

```
let newStudent = update(student, {grades: {$push: ['A']}})
```

对象 {grades: {\$push: ['A']}} 从左到右地表明了 update 函数应当：

- (1) 定位到 grades 属性(修改应该应用在“何处”)。
- (2) 将一个新值添加到数组中(应该应用“何种”修改)。

如果想要完全修改整个数组，可使用命令 \$set 来替代 \$push：

```
let newStudent = update(student, {grades: {$set: ['A', 'A', 'B']}})
```

对于对象中可以有多少层嵌套，并没有任何限制。下面回过头来看之前的机票对象，之前当我们想要使用一个不同的 arrival 信息来创建一个新对象时遇到了问题。原始的对象是：

```
let originalTicket={
  company: 'Delta',
  flightNo: '0990',
  departure: {
    airport: 'LAS',
    time: '2016-08-21T10:00:00.000Z'
  },
  arrival: {
    airport: 'MIA',
    time: '2016-08-21T14:41:10.000Z'
  },
  codeshare: [
    {company: 'GL', flightNo: '9840'},
    {company: 'TM', flightNo: '5010'}
  ]
}
```

你想要修改的信息(arrival.airport)被放在三层深的嵌套对象里面。在 React 的 update 插件中，你要做的就是使用嵌套对象的名称来描述你要做的修改：

```
let newTicket = update(originalTicket, {
  arrival: {
    airport: {$set: 'MCO'}
  }
});
```

现在，只有新的 newTicket 对象的 arrival.airport 属性被设置成了“MCO”。原始的 originalTicket 对象中的 arrival.airport 属性则仍然保持原来的值，如图 3-11 所示。

```
originalTicket
▼ Object {company: "Delta", flightNo: "0990", departure: Object, arrival: Object}
  ▼ arrival: Object
    airport: "MIA"
    time: "2016-08-21T14:41:10.000Z"
newTicket
▼ Object {company: "Delta", flightNo: "5690", departure: Object, arrival: Object}
  ▼ arrival: Object
    airport: "MCO"
    time: "2016-08-21T14:41:10.000Z"
```

图 3-11 originalTicket 和 newTicket 对象不在共享同一个 arrival 嵌套对象

1. 数组索引

另一种指定要在何处进行修改的方法，是使用数组索引。例如，如果你想要修改第一个 codeshare 对象(对象在 elopement 数组中的索引为 0)：

```
let newTicket = update(originalTicket,{
  codeshare: {
    0: { $set: {company:'AZ', flightNo:'7320'} }
  }
});
```

图 3-12 展现了 newTicket 对象的数组中包含了和原始对象中不同的值。

```
originalTicket
▼ Object {company: "Delta", flightNo: "0990", departure: Object, arrival: Object, codeshare: Array[2]}
  ► arrival: Object
  ▼ codeshare: Array[2]
    ▼ 0: Object
      company: "GL"
      flightNo: "9840"
newTicket
▼ Object {company: "Delta", flightNo: "0990", departure: Object, arrival: Object, codeshare: Array[2]}
  ► arrival: Object
  ▼ codeshare: Array[2]
    ▼ 0: Object
      company: "AZ"
      flightNo: "7320"
```

图 3-12 使用 React 的不变性助手，通过数组索引来修改对象

2. 可用的指令

表 3-4 中显示了可以用来指定进行何种数据修改的可用命令。



表 3-4 React 不变性助手的命令

命令	描述
\$push	类似于 Array 的 push 函数，它向一个数组的尾部添加一个或多个元素。例如： <pre>let initialArray = [1, 2, 3]; let newArray = update(initialArray, {\$push: [4]}); // =&gt; [1, 2, 3, 4]</pre>
\$unshift	类似于 Array 的 unshift 函数，它在一个数组的头部添加一个或多个元素。例如： <pre>let initialArray = [1, 2, 3]; let newArray = update(initialArray, {\$unshift: [0]}); // =&gt; [0, 1, 2, 3]</pre>
\$splice	类似于 Array 的 splice 函数，它通过从数组中移除元素且/或向数组中添加元素，来修改一个数组的内容。和 Array.splice 主要的语法区别是你需要提供一个元素为数组的数组来作为参数，作为参数的数组中的每一个数组包含了对数组进行 splice 操作的参数。例如： <pre>let initial Array = [1, 2, 'a']; let newArray = update(initialArray, {\$splice: [[2,1,3,4]]}); // =&gt; [1, 2, 3, 4]</pre>
\$set	完整地替换掉整个目标
\$merge	将给定对象的键合并到目标对象中。例如： <pre>let ob. = {a: 5, b: 3}; let newObj = update(obj, {\$merge: {b: 6, c: 7}}); // =&gt; {a: 5, b: 6, c: 7}</pre>
\$apply	将当前的值传给一个函数，在函数中对传入的值进行修改，然后使用函数的返回值作为结果。例如： <pre>let obj = {a: 5, b: 3}; let newObj = update(obj, {b: {\$apply: (value) =&gt; value*2 }}); // =&gt; {a: 5, b: 6}</pre>

### 3.5 看板应用：添加一点复杂性

基于学到的新的有关组件组合和状态管理的知识，你将把看板应用连接到一个外部 API。你将从服务器上获取所有的应用程序数据，并完成一些对任务的处理(删除、创建和切换)。

#### 3.5.1 从外部 API 获取初始的卡片数据

你将首先在组件层级的顶层创建一个新组件。这个容器组件将用来进行数据的获取

和保存。创建一个名为 KanbanBoardContainer.js 的新文件，文件中将包含 React 组件的基本结构(如代码清单 3-14 所示)。

代码清单 3-14: 新的 KanbanBoardContainer.js 文件

```
import React, { Component } from 'react';
import KanbanBoard from './KanbanBoard';

class KanbanBoardContainer extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      cards: [],
    };
  }

  render() {
    return <KanbanBoard cards={cards} />
  }
}

export default KanbanBoardContainer;
```

接下来，你要从看板应用的 API 服务器上获取数据。和你在本章前面部分所做的一样，你将使用最新版本的浏览器所支持的 `window.fetch` 函数进行数据获取操作。要确保你的应用能在其他浏览器上运行，从 npm 安装 `fetch polyfill`，将它保存为项目的依赖项：

```
npm install --save whatwg-fetch
```

为方便起见，可通过 <http://kanbanapi.pro-ract.com> 访问一个用于测试的在线 API。

如果想要在本地运行 API 服务器，你可从 [www.apress.com](http://www.apress.com) 或者本书的 github 页 <https://github.com/pro-react>，下载看板 API 服务器。

使用位于 [kanbanapi.pro-ract.com](http://kanbanapi.pro-ract.com) 的在线 API 和本地的 API 服务器的唯一区别，是如果你要用在线 API，就需要传递一个身份验证头信息(这样服务器才可以验证你的身份，并提供给你自己的卡片和任务数据)。身份验证可以是用来唯一标识你的应用或者你自己的任何字符串(例如可以是一组字符的任意组合，或是你的 email 地址)。不管你用哪种 API，在你第一次访问时，都会有一组标准的卡片和任务数据，来方便你立即开始进行测试。

#### 注意：

位于 [kanbanapi.pro-react.com](http://kanbanapi.pro-react.com) 的在线看板 Rest API 仅用于教学目的。所以，任何保存的信息都会在访问后的 24 小时之后被清除。

同时，请不要在 [kanbanapi.pro-react.com](http://kanbanapi.pro-react.com) 服务器上存储任何敏感信息。虽然服务器应用了标准的安全防护，但它不是用来存放私人数据的。

在线 API 的使用条款位于 <http://kanbanapi.pro-react.com/terms>。

下面开始在 KanbanBoardContainer 组件中为应用程序获取初始的数据，如代码清单 3-15 所示。注意，你为 fetch 命令添加了自定义的头信息，来确保服务器可正确地响应请求。

代码清单 3-15：获取数据的代码

```
import React, { Component } from 'react';
import KanbanBoard from './KanbanBoard';
import 'whatwg-fetch';

// If you're running the server locally, the URL will be, by default,
// localhost:3000
// Also, the local server doesn't need an authorization header.
const API_URL = 'http://kanbanapi.pro-react.com';
const API_HEADERS = {
  'Content-Type': 'application/json',
  Authorization: 'any-string-you-like' // The Authorization is not needed
  for local server
};

class KanbanBoardContainer extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      cards: []
    };
  }

  componentDidMount() {
    fetch(API_URL + '/cards', {headers: API_HEADERS})
      .then((response) => response.json())
      .then((responseData) => {
        this.setState({cards: responseData});
      })
      .catch((error) => {
        console.log('Error fetching and parsing data', error);
      });
  }

  render() {
    return <KanbanBoard cards={this.state.cards} />
  }
}

export default KanbanBoardContainer;
```

你创建了一个新的容器组件来获取远程的数据，然后传递给相应的单纯组件。你需要做的就是修改原始的 App.js 文件，来渲染新的 KanbanBoardContainer，而非直接渲染

KanbanBoard:

```
import React from 'react';
import {render} from 'react-dom';
import KanbanBoardContainer from './KanbanBoardContainer';

render(<KanbanBoardContainer />, document.getElementById('root'));
```

如果你现在进行测试，应用程序会看起来和之前没两样。看板应用现在和之前唯一的区别就是它现在使用了动态数据，而不再是将数据硬编码在代码中了。

### 3.5.2 将任务回调以 props 传递

现在让我们创建三个函数来包装与任务相关的操作：`addTask`、`deleteTask` 和 `toggleTask`。既然任务属于卡片，所有这三个函数都需要接收 `cardId` 作为参数。`addTask` 函数会接收新任务的文本，而 `deleteTask` 和 `toggleTask` 函数则应当接收 `taskId` 和 `taskIndex`(在卡片中任务数组里面的索引)。你将这三个函数作为 props 向下传递给整个底层的组件。

有一个可以节省一些敲键盘时间的技巧，就是不需要为每个新的函数创建一个 props，而可以创建一个对象，来引用那三个函数，然后将这个对象作为单个 props 传递。代码如代码清单 3-16 所示。

代码清单 3-16：对任务进行操作的新函数

```
class KanbanBoardContainer extends Component {
  constructor() {...}
  componentDidMount() {...}

  addTask(cardId, taskName){

  }

  deleteTask(cardId, taskId, taskIndex){

  }

  toggleTask(cardId, taskId, taskIndex){

  }

  render() {
    return (
      <KanbanBoard cards={this.state.cards}
        taskCallbacks={{
          toggle: this.toggleTask.bind(this),
          delete: this.deleteTask.bind(this),
          add: this.addTask.bind(this) }} />
    );
  }
}
```

```

    )
  }
}

```

现在有一些繁杂的工作需要完成：所有位于顶层组件和 `CheckList` 组件之间的组件（也即 `KanbaoBoard`、`List` 和 `Card` 组件）都必须从父组件接收 `taskCallbacks` props，然后将它作为一个 props 传递给子组件。尽管看起来确实是一项乏味的工作，但这将使得组件之间的数据流动变得非常清晰易懂。代码清单 3-17、代码清单 3-18 和代码清单 3-19 显示了这些组件更新后的代码。

代码清单 3-17: `KanbaoBoard` 组件接收并传递 `taskCallbacks` props

```

class KanbanBoard extends Component {
  render() {
    return (
      <div className="app">

        <List title="To Do" taskCallbacks={this.props.taskCallbacks}
          cards={
            this.props.cards.filter((card) => card.status === "todo")
          } />

        <List title="In Progress" taskCallbacks={this.props.taskCallbacks}
          cards={
            this.props.cards.filter((card) => card.status == "in-progress")
          } />

        <List title="Done" taskCallbacks={this.props.taskCallbacks}
          cards={
            this.props.cards.filter((card) => card.status == "done")
          } />

      </div>
    )
  }
}

KanbanBoard.propTypes = {
  cards: PropTypes.arrayOf(PropTypes.object),
  taskCallbacks: PropTypes.object
}

```

代码清单 3-18: `List` 组件接收并传递 `taskCallbacks` props

```

class List extends Component {
  render() {

    let cards = this.props.cards.map((card) => {
      return <Card key={card.id} taskCallbacks={this.props.taskCallbacks}
        {...card} />
    })
  }
}

```



```

    });
    return (...);
  }
}
List.propTypes = {
  title: PropTypes.string.isRequired,
  cards: PropTypes.arrayOf(PropTypes.object),
  taskCallbacks: PropTypes.object,
}

```

在代码清单 3-18 中，值得一提的是在向 Card 组件传递 props 时，展开操作符(spread operator)的使用让我们减少了一些敲键盘的时间。要了解有关展开操作符的更多内容，请参考在线附录。

代码清单 3-19: Card 组件接收和传递 taskCallbacks props

```

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}
  render() {
    let cardDetails;
    if (this.state.showDetails) {
      cardDetails = (
        <div className="card_details">
          <span dangerouslySetInnerHTML={{__
            html:marked(this.props.description)}} />
          <CheckList cardId={this.props.id}
            tasks={this.props.tasks}
            taskCallbacks={this.props.taskCallbacks} />
        </div>
      );
    }
    let sideColor = {...}
    return (...);
  }
}
Card.propTypes = {
  id: PropTypes.number,
  title: titlePropType,
  description: PropTypes.string,
  color: PropTypes.string,
  tasks: PropTypes.array,
  taskCallbacks: PropTypes.object,
}

```

最后，在 CheckList 组件中，你将使用 taskCallbacks.taskCallbacks.delete 和 taskCallbacks.toggle 函数，它们可以直接关联到元素的事件处理程序上：

```

class CheckList extends Component {
  render() {
    let tasks = this.props.tasks.map((task, taskIndex) => (
      <li key={task.id} className="checklist__task">
        <input type="checkbox" checked={task.done} onChange={
          this.props.taskCallbacks.toggle.bind(null, this.props.cardId,
            task.id, taskIndex)
        } />
        {task.name}{' '}
        <a href="#" className="checklist__task--remove" onClick={
          this.props.taskCallbacks.delete.bind(null, this.props.cardId,
            task.id, taskIndex)
        } />
      </li>
    ));
    return (...);
  }
}

```

但是如果要新增一个任务，你还需要在调用 `taskCallbacks.add` 回调之前，在组件里面进行一些预处理操作。这样做有两个原因：检查用户是否按下了回车键，以及在调用回调函数之后清除输入文本框的内容：

```

class CheckList extends Component {
  checkInputKeyPress(evt) {
    if(evt.key === 'Enter') {
      this.props.taskCallbacks.add(this.props.cardId, evt.target.value);
      evt.target.value = '';
    }
  }

  render() {
    let tasks = this.props.tasks.map((task, taskIndex) => (...));

    return (
      <div className="checklist">
        <ul>{tasks}</ul>
        <input type="text"
          className="checklist--add-task"
          placeholder="Type then hit Enter to add a task"
          onKeyDown={this.checkInputKeyPress.bind(this)} />
      </div>
    );
  }
}

```

CheckList 组件的完整代码如代码清单 3-20 所示。

代码清单 3-20: 完整的 CheckList 组件代码, 接收并调用了所有的任务回调函数

```
import React, { Component, PropTypes } from 'react';

class CheckList extends Component {
  checkInputKeyPress(evt) {
    if (evt.key === 'Enter') {
      this.props.taskCallbacks.add(this.props.cardId, evt.target.value)
      evt.target.value = '';
    }
  }

  render() {
    let tasks = this.props.tasks.map((task, taskIndex) => (
      <li key={task.id} className="checklist__task">
        <input type="checkbox" checked={task.done} onChange={
          this.props.taskCallbacks.toggle.bind(null, this.props.cardId,
            task.id, taskIndex)
        } />
        {task.name}{' '}
        <a href="#" className="checklist__task--remove" onClick={
          this.props.taskCallbacks.delete.bind(null, this.props.cardId,
            task.id, taskIndex)
        } />
      </li>
    ));

    return (
      <div className="checklist">
        <ul>{tasks}</ul>
        <input type="text"
          className="checklist--add-task"
          placeholder="Type then hit Enter to add a task"
          onKeyDown={this.checkInputKeyPress.bind(this)} />
      </div>
    )
  }
}

CheckList.propTypes = {
  cardId: PropTypes.number,
  taskCallbacks: PropTypes.object,
  tasks: PropTypes.array
};

export default CheckList;
```

### 3.5.3 处理任务数据

在这一节, 你要在 KanbanContainer 组件的 state 中实际处理任务数据, 并

将所有更改通过 API 保存到服务器上。在所有三个函数中(deleteTask、toggleTask 和 addTask)，你需要确保不直接访问 state，所以你将使用 React 的不变性助手。别忘了通过运行 `npm install --save react-addons-update` 来安装它们。

现在还有一个问题：既然你在 KanbanList 组件中筛选了卡片，你不能再访问它们的原始索引(在使用不变性助手时，需要使用它们的索引)。所以你可以使用新的 `findIndex()` 数组函数，对每个元素进行测试，然后返回满足测试条件的元素的索引。

### 注意：

在撰写本书时，只有 Chrome 和 Firefox 浏览器支持新的 `array.prototype.find` 和 `array.prototype.findIndex` 函数，所以要确保你安装了 `babel-polyfill`：

```
npm install --save babel-polyfill
```

然后，在你的代码文件中，使用以下语句导入它：

```
import 'babel-polyfill'
```

下面开始编写函数，首先是 `deleteTask` 函数。我们先通过 ID 来查找需要的卡片的索引。然后通过使用不变性助手来创建一个不包含被删除任务的新可变对象。最后，为对象调用 `setState`，并使用 `fetch` 函数来通知服务器有关数据的变化。

```
deleteTask(cardId, taskId, taskIndex){
  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((card)=>card.id == cardId);

  // Create a new object without the task
  let nextState = update(this.state.cards, {
    [cardIndex]: {
      tasks: {$splice: [[taskIndex,1]] }
    }
  });

  // set the component state to the mutated object
  this.setState({cards:nextState});

  // Call the API to remove the task on the server
  fetch(`${API_URL}/cards/${cardId}/tasks/${taskId}`, {
    method: 'delete',
    headers: API_HEADERS
  });
}
```

切换一个任务基本上也是执行相似的操作，但我们并不从数组中删除元素，而是上移到任务对象的 `done` 属性，然后使用一个函数直接将它的值修改成反向的值：

```
toggleTask(cardId, taskId, taskIndex){
  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((card)=>card.id == cardId);
```

```

// Save a reference to the task's 'done' value
let newDoneValue;
// Using the $apply command, you will change the done value to its opposite
let nextState = update(this.state.cards, {
  [cardIndex]: {
    tasks: {
      [taskIndex]: {
        done: { $apply: (done) => {
          newDoneValue = !done
          return newDoneValue;
        }
      }
    }
  }
});

// set the component state to the mutated object
this.setState({cards:nextState});

// Call the API to toggle the task on the server
fetch(`${API_URL}/cards/${cardId}/tasks/${taskId}`, {
  method: 'put',
  headers: API_HEADERS,
  body: JSON.stringify({done:newDoneValue})
});
}

```

你可能已经想到了，添加一个新任务也与此类似。唯一需要注意的是由于所有任务都需要一个 ID，你必须为任务生成一个临时的 ID，直到它被保存到服务器上，然后服务器会返回这个任务最终的 ID。然后你必须更新任务对象的 ID。临时的 ID 可以使用像当前时间的毫秒数这样简单的值：

```

addTask(cardId, taskName){
  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((card)=>card.id == cardId);

  // Create a new task with the given name and a temporary ID
  let newTask = {id:Date.now(), name:taskName, done:false};

  // Create a new object and push the new task to the array of tasks
  let nextState = update(this.state.cards, {
    [cardIndex]: {
      tasks: {push: [newTask] }
    }
  });

  // set the component state to the mutated object
  this.setState({cards:nextState});
}

```



```
// Call the API to add the task on the server
fetch(`${API_URL}/cards/${cardId}/tasks`, {
  method: 'post',
  headers: API_HEADERS,
  body: JSON.stringify(newTask)
})
.then((response) => response.json())
.then((responseData) => {
  // When the server returns the definitive ID
  // used for the new Task on the server, update it on React
  newTask.id=responseData.id
  this.setState({cards:nextState});
});
}
```

### 3.5.4 基本的乐观更新回滚

你也许已经注意到，对所有在 UI 上的更新都是乐观的，也就是说，还没有等到服务器真正返回响应指出更改是否已经被保存，就已经更新了 UI。乐观更新对于用户体验很重要：当用户和一个在线 app 进行交互时，他们不想等待。他们不关心是否他们的任务需要被保存到一个远程数据库中。所有操作都需要实时完成。但是如果服务器保存数据失败该怎么办呢？你需要做一些新尝试、回滚 UI 上的更新、提示用户或执行其他操作。

乐观更新和回滚不是一件简单任务，它可以被展开详细讲述，但现在将简单介绍基本的回滚操作，使用不可变数据结构的一个额外好处是：你可以保存旧 state 对象的一个引用，然后在发生问题时回滚到这个旧对象上。

对于所有三个任务回调，代码都是相同的。首先，保存一份组件原始 state 的引用：

```
// Keep a reference to the original state prior to the mutations
// in case you need to revert the optimistic changes in the UI
let prevState = this.state;
```

接下来，如果 fetch 命令失败或服务器返回一个命令操作未成功的状态，就使用 setState 来回滚到原来的 state：

```
fetch(..., {...})
.then((response) => {
  if(!response.ok){
    // Throw an error if server response wasn't 'ok'
    // so you can revert back the optimistic changes
    // made to the UI.
    throw new Error("Server response wasn't OK")
  }
})
.catch((error) => {
  console.error("Fetch error:",error)
```

```

    this.setState(prevState);
  });

```

为进行测试, 你可以关闭本地 API 服务器(如果你使用的是在线 API, 则断开网络连接), 然后尝试对任务进行任意修改。

KanbanBoardContainer 组件的完整代码如代码清单 3-21 所示。

代码清单 3-21: 包含了任务操作函数的 KanbanBoardContainer 组件完整代码

```

import React, { Component } from 'react';
import update from 'react-addons-update';
import KanbanBoard from './KanbanBoard';
// Polyfills
import 'babel-polyfill';
import 'whatwg-fetch';

const API_URL = 'http://kanbanapi.pro-react.com';
const API_HEADERS = {
  'Content-Type': 'application/json',
  Authorization: 'any-string-you-like' // The Authorization is not needed
  for local server
};

class KanbanBoardContainer extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      cards: [],
    };
  }

  componentDidMount() {
    fetch(`${API_URL}/cards`, {headers: API_HEADERS})
      .then((response) => response.json())
      .then((responseData) => {
        this.setState({
          cards: responseData
        });

        window.state = this.state;
      });
  }

  addTask(cardId, taskName) {
    // Keep a reference to the original state prior to the mutations
    // in case you need to revert the optimistic changes in the UI
    let prevState = this.state;

    // Find the index of the card

```

```

let cardIndex = this.state.cards.findIndex((card)=>card.id == cardId);

// Create a new task with the given name and a temporary ID
let newTask = {id:Date.now(), name:taskName, done:false};
// Create a new object and push the new task to the array of tasks
let nextState = update(this.state.cards, {
  [cardIndex]: {
    tasks: {$push: [newTask]}
  }
});

// set the component state to the mutated object
this.setState({cards:nextState});

// Call the API to add the task on the server
fetch(`${API_URL}/cards/${cardId}/tasks`, {
  method: 'post',
  headers: API_HEADERS,
  body: JSON.stringify(newTask)
})
.then((response) => {
  if(response.ok){
    return response.json()
  } else {
    // Throw an error if server response wasn't 'ok'
    // so you can revert back the optimistic changes
    // made to the UI.
    throw new Error("Server response wasn't OK")
  }
})
.then((responseData) => {
  // When the server returns the definitive ID
  // used for the new Task on the server, update it on React
  newTask.id=responseData.id
  this.setState({cards:nextState});
})
.catch((error) => {
  this.setState(prevState);
});
}

deleteTask(cardId, taskId, taskIndex){
  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((card)=>card.id == cardId);

  // Keep a reference to the original state prior to the mutations
  // in case you need to revert the optimistic changes in the UI
  let prevState = this.state;

```

```

// Create a new object without the task
let nextState = update(this.state.cards, {
  [cardIndex]: {
    tasks: {$splice: [[taskIndex,1]]}
  }
});

// set the component state to the mutated object
this.setState({cards:nextState});

// Call the API to remove the task on the server
fetch(`${API_URL}/cards/${cardId}/tasks/${taskId}`, {
  method: 'delete',
  headers: API_HEADERS
})
.then((response) => {
  if(!response.ok){
    // Throw an error if server response wasn't 'ok'
    // so you can revert back the optimistic changes
    // made to the UI.
    throw new Error("Server response wasn't OK")
  }
})
.catch((error) => {
  console.error("Fetch error:",error)
  this.setState(prevState);
});
}

toggleTask(cardId, taskId, taskIndex){
  // Keep a reference to the original state prior to the mutations
  // in case you need to revert the optimistic changes in the UI
  let prevState = this.state;

  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((card)=>card.id == cardId);
  // Save a reference to the task's 'done' value
  let newDoneValue;
  // Using the $apply command, you will change the done value to its opposite,
  let nextState = update(
    this.state.cards, {
      [cardIndex]: {
        tasks: {
          [taskIndex]: {
            done: { $apply: (done) => {
              newDoneValue = !done
              return newDoneValue;
            }
          }
        }
      }
    }
  )
}

```

```

    }
  }
});

// set the component state to the mutated object
this.setState({cards:nextState});

// Call the API to toggle the task on the server
fetch(`${API_URL}/cards/${cardId}/tasks/${taskId}`, {
  method: 'put',
  headers: API_HEADERS,
  body: JSON.stringify({done:newDoneValue})
})
.then((response) => {
  if(!response.ok){
    // Throw an error if server response wasn't 'ok'
    // so you can revert back the optimistic changes
    // made to the UI.
    throw new Error("Server response wasn't OK")
  }
})
.catch((error) => {
  console.error("Fetch error:",error)
  this.setState(prevState);
});
}

render() {
  return (
    <KanbanBoard cards={this.state.cards} taskCallbacks={{
      toggle: this.toggleTask.bind(this),
      delete: this.deleteTask.bind(this),
      add: this.addTask.bind(this) }} />
  )
}
}

export default KanbanBoardContainer;

```

## 3.6 本章小结

在本章中，你学习了如何在 React 中构建复杂的 UI 界面。了解到在一个 React 应用程序中，数据总是沿着一个方向，从父组件向子组件流动。为进行数据传递，一个父组件可将一个回调函数作为 props 向下传递给子组件，这样子组件就可以使用接收到的回调函数将数据传回上层。

你还了解到如果你将组件分成两类：有状态组件(包含有内部的 state)和单纯组件(没



有内部的 state，只显示通过 props 接收到的数据)，那么就会更容易地复用组件。将你的应用程序按照这种方式进行构建是一项最佳实践，这样应用程序就会包含更少有状态组件(有状态组件通常位于应用程序组件层级的顶层)和更多单纯组件。

最后，你了解到为何将组件的 state 看成不可变数据是很重要的，总是应当使用 this.setState 来修改 state(你学会了如何使用 React 的不变性助手来生成基于 this.state 的修改后的浅表副本)。

## 复杂交互

按今天的应用程序标准看,仅拥有所需的功能、快捷的加载速度、不错的感知性能是不够的。用户界面还必须精致、流畅,并包含一些复杂交互,比如元素动画效果以及拖放交互。

### 4.1 React 中的动画

通过使用高层级的 `ReactCSSTransitionGroup`(插件模块的一部分),`React` 提供了一组默认方法用于处理动画。`ReactCSSTransitionGroup` 并非一个完整动画库,它并不包含插值运算、时间轴管理、链式动画等特性,但它帮助我们在 `React` 中集成了 `CSS` 的过渡动画(`Transition`),允许你在组件添加到 `DOM` 节点或从 `DOM` 节点中删除时触发 `CSS` 动画。`CSS` 中的 `transition` 和 `animation` 动画是浏览器标准机制的一部分,可通过插值方式从一种 `CSS` 样式平滑过渡到另一种。

在接下来的两节中,你会简要了解到 `CSS` 动画是如何运作的,以及如何使用 `ReactCSSTransitionGroup` 来实现组件动画。

#### 4.1.1 CSS 过渡和动画基础

为使用 `ReactCSSTransitionGroup`,你需要先熟悉 `CSS` 中的过渡和动画设置,以及了解如何使用 `JavaScript` 来触发这些动画。在介绍 `React` 组件动画集成之前,先来简要介绍一下这个主题。如果你已经掌握了 `CSS` 动画,尽可以直接跳到下一节中和 `React` 相关的部分。

`CSS` 中的动画分为两类:过渡(`transition`)和关键帧动画(`keyframe animation`)。

- `CSS` 过渡是在两个不同状态之间进行插值实现动画效果,它包含一个开始状态和一个结束状态。
- `CSS` 关键帧动画可进行更复杂的控制,使用关键帧的形式在开始和结束状态之间设置中间步骤。

##### 1. `CSS` 过渡

`CSS` 过渡提供了一种在 `CSS` 属性之间实现动画(更准确地说,是插值)的方式。例如,

你把一个元素的颜色从灰色变成红色，通常情况下，这个变化是瞬间生效的。如果启用了 CSS 过渡效果，这个变化就可以在指定的一段时间内平滑完成。

通过 `transition` 属性来控制 CSS 过渡效果。它会告诉浏览器，在一段时间内，对选择器中包含的属性进行插值运算，从而实现动画效果。`transition` 属性最多可接收四个特性：

- 需要进行动画处理的元素属性名称(如 `color` 或 `width`)。如果省略该参数，所有可实现动画效果的属性都会被进行过渡处理。
- 动画的持续时间。
- 可选的时间方法用于控制动画的加速曲线(如 `ease-in` 和 `ease-out`)。
- 可选的延迟时间(在动画开始之前)。

下面创建一个按钮形状的 HTML 链接，当鼠标移动到上面时改变其背景色。在代码清单 4-1 中，注意 `.button` 和 `.button:hover` 选择器，它们的 `background-color` 和 `box-shadow` 属性包含了不同的值，同样需要注意 `transition` 属性中定义的动画持续时间。图 4-1 展示了在鼠标移到按钮上方(hover)时，按钮的动画显示过程。

代码清单 4-1: CSS transition 属性

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hover Transition</title>
    <style media="screen">
      a{
        font-family: Helvetica, Arial, sans-serif;
        text-decoration:none;
        color:#ffffff;
      }

      .button{
        padding:0.75rem1rem;
        border-radius:0.3rem;
        box-shadow:0;
        background-color:#bbbbbb;
      }
      .button:hover{
        background-color:#ee2222;
        box-shadow:04px#990000;
        transition:0.5s;
      }
    </style>
  </head>
  <body>
    <a href="#" class="button"> Hover Me!</div>
  </body>
</html>
```

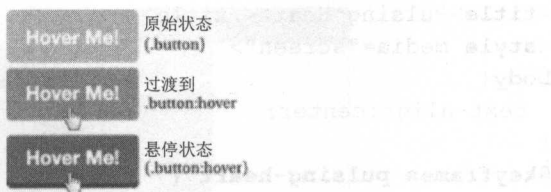


图 4-1 hover 动画过程的演示

### 注意前缀

在本书撰写时，一些基于 WebKit 内核的浏览器依然要求在 CSS 属性名前面使用前缀，包括动画(animation)、关键帧(keyframe)和过渡(transition)。在它们完全兼容标准版本之前，你可能需要在代码中同时包含不带前缀和带有前缀的版本。

例如，按钮在 hover 状态时的代码应该包括：

```
.button:hover{
  background-color:#ee2222;
  box-shadow:04px#990000;
  webkit-transition:0.5s;
  transition:0.5s;
}
```

为简单起见，本书中的示例代码使用不带前缀的版本。

## 2. 关键帧动画

基于过渡效果的动画只能控制两点之间的动画效果：开始状态和结束状态。所有中间状态都是由浏览器进行插值计算得到的。另一种创建 CSS 动画的方法是 keyframe 属性，它可以让你更精确地控制动画序列的中间状态，而不是让浏览器自动完成所有处理。

使用关键帧动画时，你需要在一个单独的 CSS 块中通过 @keyframe 规则和名称来定义你的动画步骤，例如：

```
@keyframes pulsing-heart {
  0%{transform:none;}
  50% {transform: scale(1.4);}
  100% {transform:none;}
}
```

上面的代码段中是一组名为 pulsing-heart 的关键帧。它定义了三个关键帧：一个在动画最开始时(定义为 0%)；一个在动画的中间(50%)；一个在动画的结尾。

在后续任何一个样式定义中，都可通过 animation 属性来引用预先定义的关键帧。animation 属性接收关键帧的名称、动画持续时间和其他可选的配置(如重复次数)。下例创建一个动画，在鼠标移动到目标上时实现心跳效果。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
```

```

<title>Pulsing Heart</title>
<style media="screen">
body{
  text-align:center;
}
@keyframes pulsing-heart {
  0%{transform:none;}
  50% {transform: scale(1.4);}
  100% {transform:none;}
}
.heart{
  font-size:10rem;
  color:#FF0000;
}

.heart:hover{
  animation: pulsing-heart .5s infinite;
  transform-origin:center;
}
</style>
</head>
<body>
  <div>
    <div class="heart">&hearts;</div>
  </div>
</body>
</html>

```

### 3. 编程启动 CSS 过渡和动画

由于 CSS 选择器中的伪类只能涵盖最基本的交互场景，你可能需要使用 JavaScript 来更灵活地控制 CSS 过渡和关键帧动画的触发时机。这通常通过交换 class 的方式来实现：你需要针对相同的元素创建两个独立的 class，包含不同的属性值。HTML 元素使用其中一个 class 作为初始值，通过 JavaScript 你可以动态移除原 class 引用、并添加上新的 class，从而触发 CSS 的动画效果。

下面用一个非常基础的原型来尝试：一个通过页面标题中的“汉堡包按钮”触发的侧边栏菜单，如图 4-2 所示：

首先通过一个 CSS class 来定义基本的侧边栏样式：

```

/* Sidebar default style */
.sidebar{
  background-color:#eee;
  box-shadow:1px03px#888888;
  position:absolute;
  width:15rem;
  height:100%;
}

```



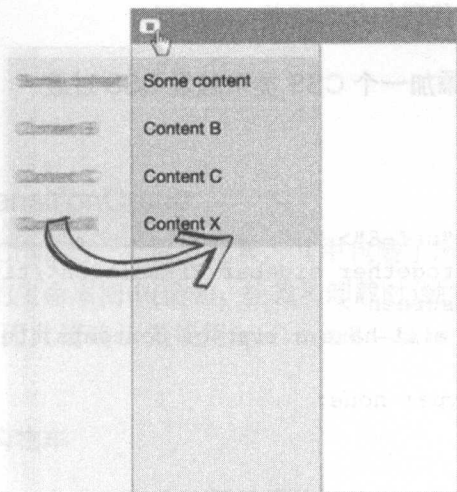


图 4-2 通过单击按钮打开侧边栏

接下来创建两个 class，它们包含相同的属性，但拥有不同的值。第一个 class(.sidebar-transition)将侧边栏的透明度设置为 0(透明)并将其位置设置在屏幕边界之外，而.sidebar-transition-active则将其透明度设置为 1(可见)并将其位置设置在屏幕之内。注意sidebar-transition-active类同样定义了transition属性，其动画持续时间为 0.5 秒。

```
.sidebar-transition{
  opacity:0;
  left:-15rem;
}
.sidebar-transition-active{
  opacity:1;
  left:0;
  transition: ease-in-out 0.5s;
}
```

在HTML代码中，侧边栏在声明时只使用.sidebar-transition类，不使用.sidebar-transition-active(所以开始时侧边栏是隐藏的)：

```
<div class='sidebar sidebar-transition'>
  <ul>
    <li>Some content</li>
    <li>Content B</li>
    ...
    <li>Concent X</li>
  </ul>
</div>
```

这里不会使用任何 React 相关的库。为触发这个最基本的示例，你将使用一段内嵌在 HTML 代码中的 JavaScript，这其实并不是一个最佳实践，只是作为一个原型用于演示效果。在这里 JavaScript 代码做的就是单击菜单按钮时，将.sidebar-transition-active

类添加到侧边栏上。示例代码如代码清单 4-2 所示。

代码清单 4-2: 动态添加一个 CSS 类来触发 CSS 过渡

```
<!DOCTYPEhtml>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hacked together sidebar Transition</title>
    <style media="screen">
      /* the sidebar will have a list of contents. Let's style them too */
      ul {
        list-style-type: none;
        padding:0;
      }
      li{
        padding: 15px;
        border-bottom: solid1px#eee;
        background-color: #ddd;
      }
      .sidebar{
        background-color:#eee;
        box-shadow: 1px 03px #888888;
        position:absolute;
        width:15rem;
        height:100%;
      }
      .sidebar-transition{
        opacity: 0;
        left: -15rem;
      }
      .sidebar-transition-active{
        opacity: 1;
        left: 0;
        transition: 0.5s;
      }
    </style>
  </head>
  <body>
    <header>
      <button onclick="
        document.querySelector('.sidebar').classList.add(
          'sidebar-transition-active');
      ">&#9776; </button>
      <!-- &#9776; is the HTML Entity for the ☰ utf-8 symbol (
        aka "Hamburger Menu") -->
    </header>
    <div class='sidebar sidebar-transition'>
      <ul>
```

```

    ...
  </ul>
</div>
</body>
</html>

```

### 4.1.2 ReactCSSTransitionGroup

ReactCSSTransitionGroup是一个简单元素，其中包装了所有你实现动画效果需要的元素，它会在组件特定的生命周期中(例如，挂载和卸载时)触发CSS动画和过渡效果。它是通过插件形式提供的，因此请确认先使用 `npm install --save react-addons-css-transition-group` 安装了该插件。

#### React 动画示例：购物车

作为示例，下面创建一个带有简单动画效果的购物车，你可在其中添加或删除条目。

#### 基本的应用程序设置

首先创建一个新的 React 项目(你可以使用本书的应用程序样板，它可访问如下网址：<https://github.com/pro-react/react-app-boilerplate>)，然后编辑其中的主 JavaScript 文件，创建 AnimatedShoppingList 基本结构，如代码清单 4-3 所示。

#### 代码清单 4-3: AnimatedShoppingList 组件

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';

class AnimatedShoppingList extends Component {
  constructor() {
    super(...arguments);
    // Create an "items" state pre-populated with some shopping items
    this.state = {
      items: [
        { id: 1, name: 'Milk' },
        { id: 2, name: 'Yogurt' },
        { id: 3, name: 'Orange Juice' },
      ]
    };
  }

  // Called when the user changes the input field
  handleChange(evt) {
    if (evt.key === 'Enter') {
      // Create a new item and set the current time as it's id
      let newItem = { id: Date.now(), name: evt.target.value };
      // Create a new array with the previous items plus the value the user typed
      let newItems = this.state.items.concat(newItem);

```

```

    // Clear the text field
    evt.target.value='';
    // Set the new state
    this.setState({items: newItems});
  }
}

// Called when the user Clicks on a shopping item
handleRemove(i) {
  // Create a new array without the clicked item
  var newItems = this.state.items;
  newItems.splice(i, 1);
  // Set the new state
  this.setState({items: newItems});
}

render() {
  let shoppingItems = this.state.items.map((item, i) => (
    <div key={item.id}
      className="item"
      onClick={this.handleRemove.bind(this, i)}>
        {item.name}
      </div>
  ));

  return (
    <div>
      {shoppingItems}
      <input type="text" value={this.state.newItem}
        onKeyDown={this.handleChange.bind(this)} />
    </div>
  );
}
};

render(<AnimatedShoppingList />, document.getElementById('root'));

```

在这个组件中，需要注意以下几点：

- 单击购物车中的一个条目会将其移除。
- 用户可在文本框中输入内容并按下回车键来创建新条目。
- 每个购物车条目都拥有一个 ID(你甚至可在每次创建新项目时，基于时间戳来创建一个新的 ID)。这些 ID 会作为条目的 key。对于 `ReactCSSTransitionGroup` 中的所有子项，你都必须提供 key 特性，即使其中只渲染一个条目，因为 React 会根据它来判断哪个子项是新增的、删除的或是需要保留的。

下面继续编写一些 CSS 规则来创建基本样式。目前，CSS 中没有包含任何过渡规则，我们会在下一步添加它们。

```

input {
  padding: 5px;
}

```

```

width:120px;
margin-top:10px;
}

.item{
  background-color:#efefef;
  cursor:pointer;
  display:block;
  margin-bottom:1px;
  padding:8px 12px;
  width:120px;
}

```

### 添加 ReactCSSTransitionGroup 元素

这个组件现在已经可以正常工作了，可添加、移除购物车条目。现在给条目的移入和移出加上动画效果。

ReactCSSTransitionGroup元素必须包装在你需要实现动画效果的子元素外。它可以接收三个属性：transitionName(映射到CSS中包含实际动画定义的类名)、transitionEnterTimeout和transitionLeaveTimeout(定义了动画的持续时间，以毫秒为单位)。

在你的购物车列表示例中，在render方法中，在shoppingItem变量外面插入ReactCSSTransitionGroup元素。我们会把这个过渡效果命名为example，并设置进入和离开的持续时间为300毫秒：

```

return (
  <div>
    <ReactCSSTransitionGroup transitionName="example"
      transitionEnterTimeout={300}
      transitionLeaveTimeout={300}>
      {shoppingItems}
    </ReactCSSTransitionGroup>
    <input type="text" value={this.state.newItem} onKeyDown=
      {this.handleChange.bind(this)} />
  </div>
);

```

完成这一步后，每次新的条目添加到购物车中时，React会使用额外的className:example-enter来渲染条目。紧接着，在浏览器下一个tick中，React会附加一个className:example-enter-active。这是因为CSS过渡动画的本质决定的：它需要一个类包含默认样式，触发动画时会把带有不同属性值和transition规则的另一个类添加到元素上。最终，在经过了transitionEnterTimeout定义的持续时间之后，这两个类都会被移除。

下面在CSS中添加example-enter和example-enter-active类。在这个示例项目中，过渡效果使用translateX属性来实现(它会让条目从屏幕左侧滑入)：

```

.example-enter{
  opacity:0;
  transform: translateX(-250px);
}

```



```

}
.example-enter.example-enter-active{
  opacity:1;
  transform: translateX(0);
  transition:0.3s;
}

```

图 4-3 展示了新的条目动画。

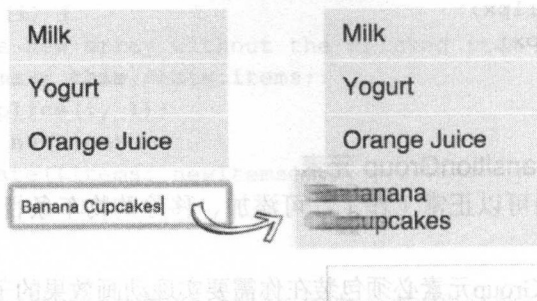


图 4-3 新条目动画

同样的机制也应用于元素从 DOM 中移除时。从购物车中移除一个条目之前，React 会向 `className` 中添加 `example-leave`，然后是 `example-leave-active`。当经过定义在 `transitionLeaveTimeout` 中的时间后，React 最终会从 DOM 中移除该元素。向 CSS 中添加如下内容来完成这个示例：

```

.example-leave{
  opacity:1;
  transform:translateX(0);
}
.example-leave.example-leave-active{
  opacity:0;
  transform:translateX(250px);
  transition:0.3s;
}

```

### 初始组件挂载时的动画

在测试示例代码时，你可能会意识到这些动画效果是在添加和移除元素时触发的，但一开始被硬编码进去的条目并不会在最开始以过渡形式出现。ReactCSSTransitionGroup 还提供了可选的属性 `transitionAppear` 用于在组件初始挂载时执行一个额外的过渡阶段。通常在组件初始挂载时是没有动画阶段的，因为 `transitionAppear` 的默认值是 `false`。在下面的示例代码中，将 `transitionAppear` 属性定义为 `true`：

```

<ReactCSSTransitionGroup transitionName="example"
  transitionEnterTimeout={300}
  transitionLeaveTimeout={300}
  transitionAppear={true}
  transitionAppearTimeout={300}>

```

```
{shoppingItems}
</ReactCSSTransitionGroup>
```

你同样需要提供额外的 CSS 类来控制元素出现的过渡动画：

```
.example-appear{
  opacity:0;
  transform: translateX(-250px);
}
.example-appear.example-appear-active{
  opacity:1;
  transform: translateX(0);
  transition: .5s;
}
```

现在，你的应用程序会包含初始元素的动画效果了。代码清单 4-4 中展示了带有动画效果的购物车列表应用程序的完整代码。

代码清单 4-4: AnimatedShoppingList 应用程序的完整源代码

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';

class AnimatedShoppingList extends Component {
  constructor(){
    super(...arguments);
    // Create an "items" state pre-populated with some shopping items
    this.state={
      items: [
        {id:1, name: 'Milk'},
        {id:2, name: 'Yogurt'},
        {id:3, name: 'Orange Juice'},
      ]
    }
  }

  // Called when the user changes the input field
  handleChange(evt) {
    if(evt.key === 'Enter'){
      // Create a new item and set the current time as it's id
      let newItem = {id:Date.now(), name:evt.target.value}
      // Create a new array with the previous items plus the value the user typed
      let newItems = this.state.items.concat(newItem);
      // Clear the text field
      evt.target.value='';
      // Set the new state
      this.setState({items: newItems});
    }
  }
}
```

```

    }

    // Called when the user Clicks on a shopping item
    handleRemove(i) {
      // Create a new array without the clicked item
      var newItems = this.state.items;
      newItems.splice(i, 1);
      // Set the new state
      this.setState({items: newItems});
    }

    render() {
      let shoppingItems = this.state.items.map((item, i) => (
        <div key={item.id} className="item"
          onClick={this.handleRemove.bind(this, i)}>
            {item.name}
          </div>
        ));

      return (
        <div>
          <ReactCSSTransitionGroup transitionName="example"
            transitionEnterTimeout={300}
            transitionLeaveTimeout={300}
            transitionAppear={true}
            transitionAppearTimeout={300}>
            {shoppingItems}
          </ReactCSSTransitionGroup>
          <input type="text" value={this.state.newItem} onKeyDown=
            {this.handleChange.bind(this)} />
          </div>
        );
      }
    };

    render(<AnimatedShoppingList />, document.getElementById('root'));
  }

```

## 4.2 拖放

在复杂用户界面中，拖放是一个十分常见的功能。它指的是你“抓住”一个对象，然后把它拖曳到不同位置。开发拖放交互是很需要技巧的。直到不久前，浏览器中才有了标准的拖放 API。即使是在现代主流浏览器中(内置支持 HTML5 的拖放 API)，不同厂商对这些 API 的实现也略有不同，而且移动设备中依然不支持拖放 API。出于这些原因，我们将使用 React DnD 这样一个拖放库，让我们能用 React 方式开发(不需要涉及 DOM、拥抱单向的数据流、使用纯数据的方式来定义拖曳源和放置目标的逻辑，以及其他一些优势)。在底层，React DnD 会使用浏览器开放的 API(比如桌面浏览器中默认的 HTML5 API)，处理各个厂商之间 API 的不一致性和怪异性，隐藏其具体的实现细节。

提示：

注意，作为一个外部库，需要使用 npm 来安装 React DnD，并将其声明为依赖项。本书示例中使用了 React DnD 2 和 HTML5 的后端支持，可通过 `npm install --save react-dnd@2.x.x react-dnd-html5-backend@1.x.x` 进行安装。

4.2.1 React DnD 实现概述

在你的 React 应用程序中，通过 React DnD 库实现拖放行为需要使用高阶组件 (higher-order components)。高阶组件是一些 JavaScript 函数，它们使用组件对象作为传入参数，并返回一些附带了额外功能的增强版组件。

React DnD 库提供了三个高阶组件，它们必须应用于你的应用程序的不同组件中，这三个高阶组件分别是 DragSource、DropTarget 和 DragDropContext。

DragSource 会返回指定组件的增强版，使组件成为一个可被拖曳的元素；DropTarget 同样会返回增强版的组件，使其有能力处理被拖放到其内部的元素；而 DragDropContext 封装了发生拖放交互行为的父元素，在交互场景背后设置了共享的拖放状态(这也是其最简单的实现方式)。

React DnD 库还提供了装饰器(decorator)模式，可使用 JavaScript 装饰器来替代高阶组件。JavaScript 装饰器目前依然处于试验阶段，还没有成为 ES 2015 规范的一部分，所以本书的示例中依然使用高阶组件的方式。

4.2.2 React DnD 实现示例

下面通过一个示例来了解这些功能是如何配合工作的。在这个示例中，你将要实现一个小吃店主题的应用程序，其中使用很多圆圈来代表不同的小吃，它们可以被拖曳到一个购物车区域中。图 4-4 展示了最终完成后的效果。

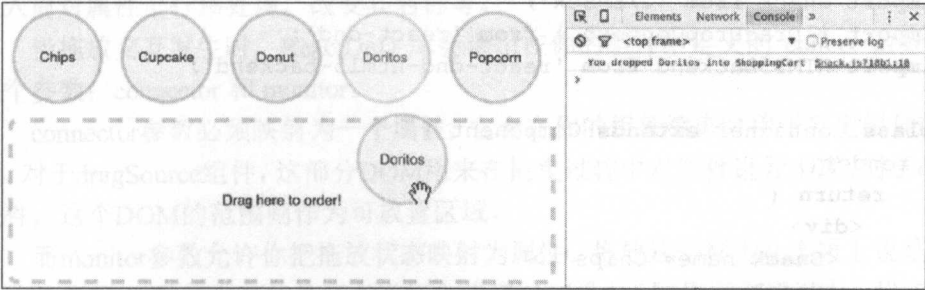


图 4-4 小吃的拖放交互

除了主 App 组件外，这个应用程序由三个组件构成：一个可被拖曳的 Snack 组件(使用 DragSource 将其增强为高阶组件)；ShoppingCart 组件(使用 DropTarget 将其增强为高阶组件)；还有一个 Container 组件，其中包含了 ShoppingCart 和多个 Snack，使用 DragDropContext 增强为高阶组件，使其能在这两者之间实现拖放功能。

因为 DragDropContext 是 React DnD 中最容易实现的部分，所以让我们自上而下开始应用程序的开发，先从 App 组件开始，随后是 Container、Snack 和 ShoppingCart。

App 组件简单易懂：它只是导入并渲染了容器的高阶组件。代码清单 4-5 展示了其源代码。

代码清单 4-5：主 App 组件

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import Container from './Container'

class App extends Component {
  render() {
    return (
      <Container />
    );
  }
}

render(<App />, document.getElementById('root'));
```

### 1. Container

接下来创建 Container 组件，所有的拖放交互都会在这里发生。其中会渲染多个 Snack 组件，它们拥有不同的名称属性；在其下方还有一个 ShoppingCart 组件。代码清单 4-6 中展示了这部分源代码。

代码清单 4-6：Container.js 文件

```
import React, { Component } from 'react';
import ShoppingCart from './ShoppingCart';
import Snack from './Snack';
import { DragDropContext } from 'react-dnd';
import HTML5Backend from 'react-dnd-html5-backend';

class Container extends Component {
  render() {
    return (
      <div>
        <Snack name='Chips' />
        <Snack name='Cupcake' />
        <Snack name='Donut' />
        <Snack name='Doritos' />
        <Snack name='Popcorn' />
        <ShoppingCart />
      </div>
    );
  }
}
```



```
export default DragDropContext (HTML5Backend) (Container);
```

需要特别注意，这个模块并没有直接导出 `Container` 组件，而在 `Container` 的基础上导出一个高阶组件，其中注入了所有与拖放相关的状态和函数。同样要注意，这里导入了 `React DnD` 的 `HTML5` 后端，如前文所述，`React` 拖放库可以支持不同的后端。

## 2. DragSource 和 DropTarget 高阶组件

接下来，我们来创建 `Snack` 和 `ShoppingCart` 组件，它们分别使用 `dragSource` 和 `dropTarget` 进行了封装增强。`dragSource` 和 `dropTarget` 都需要通过文件范例进行一些设置，稍后会解释它们。使用它们来创建高阶组件，你需要提供三个参数：`type`、`spec` 和一个 `collect` 函数。

### type

这个参数指定了组件的名称。在复杂的 UI 中，可能会出现多个不同类型的拖曳源和多个不同类型的放置目标之间的交互，所以给每个源或目标一个特定的标识是非常重要的。

### spec 对象

`spec` 对象描述了增强组件是如何响应拖曳和放置事件的。它是一个包含了若干函数的普通 JavaScript 对象，这些函数会在拖曳交互发生时被调用，例如 `beginDrag` 和 `endDrag` (对于 `DragSource` 组件) 以及 `canDrag` 和 `onDrop` (对于 `DragTarget` 组件)。

### collect 函数

`collect` 函数看上去确实很复杂，不过其实它挺简单的。下面来回顾一下，在第 3 章中，你知道了在 `React` 组件之间通过 `props` 属性互相传递信息。对于 `React DnD` 来说其实也是一样的：`dragSource` 和 `dropTarget` 的封装都向其内部组件中提供了属性注入。

`React DnD` 并非直接在你的组件中注入所有 `props` 属性，而通过 `collect` 函数让你来控制哪些属性需要进行注入以及如何进行注入。这种方式给你提供了强大能力，包括在注入前对属性进行预处理、改变其名称等。

当拖放交互发生时，`ReactDnD` 库会调用在你的组件中定义的 `collect` 函数，并传入两个参数：`connector` 和 `monitor`。

`connector` 参数必须映射为一个属性，它会在你的组件渲染时用于界定组件 DOM 的范围。对于 `dragSource` 组件，这部分 DOM 用来在拖曳过程中对组件进行呈现。对于 `dropTarget` 组件，这个 DOM 的范围则作为可放置区域。

而 `monitor` 参数允许你把拖放状态映射为属性。拖放这个行为从本质上说是一个带有状态的操作(它的状态可能是正在进行中或正在闲置中。如果正在进行中，状态中还会包括当前拖曳的组件类型和当前的条目。如果用户正在拖曳项目，状态中还可能会包括它是否正位于一个放置对象的上方，诸如此类)。使用 `monitor` 参数你可以创建诸如 `isDragging` 或者 `canDrop` 的属性，如果需要根据不同的值来渲染不同的内容，它们会是非常有用的(比如在拖曳过程中使用不同的文本或 CSS 属性来渲染相应的元素)。



### 3. ShoppingCart 组件

下面看看它是如何使用的。我们从 ShoppingCart 组件的基本框架开始，先不用 dropTarget 对其进行封装。代码清单 4-7 中展示了这部分源代码。

代码清单 4-7: ShoppingCart 组件的基本框架

```
import React, { PropTypes, Component } from 'react';
import { DropTarget } from 'react-dnd';

class ShoppingCart extends Component {
  render() {

    const style = {
      backgroundColor: '#FFFFFF'
    };

    return (
      <div className='shopping-cart' style={style}>
        Drag here to order!
      </div>
    );
  }
}
```

如你所见，这基本上是一个 render 函数，其中返回了一个 div。还包含一些内嵌的 CSS 样式，将 backgroundColor 属性设置为 white。

然后，实现一个 spec 对象。你应该还记得，spec 对象描述了放置目标是如何响应拖曳和放置事件的。这里，你只需要响应放置事件(它会在 dragSource 放置时被调用)。代码清单 4-8 展示了更新的代码，为简单起见，其中省略了部分代码。

代码清单 4-8: ShoppingCart 组件中 spec 对象的实现

```
import React, { PropTypes, Component } from 'react';
import { DropTarget } from 'react-dnd';

// ShoppingCart DND Spec
// "A plain object implementing the drop target specification"
//
// - DropTarget Methods (All optional)
// - drop: Called when a compatible item is dropped.
// - hover: Called when an item is hovered over the component.
// - canDrop: Use it to specify whether the drop target is able to accept
//           the item.
const ShoppingCartSpec = {
  drop() {
    return { name: 'ShoppingCart' };
  }
}
```

```

    }
  };

  class ShoppingCart extends Component {
    render() {...}
  }

```

在这个示例中，当放置事件发生时你只需要返回一个字符串，这段文本会在后面的 Snack 组件中使用。

接下来，你需要实现 `collect` 函数，在函数中把 React DnD 中的 `connector` 和状态映射为组件的 `props` 属性。这里你将向组件注入三个属性：`connectDropTarget` (需要 `connector`)、`isOver` 和 `canDrop`。

单独的 `collect` 函数如下所示：

```

// ShoppingCart DropTarget - collect
//
// - connect: An instance of DropTargetConnector.
//   You use it to assign the drop target role to a DOM node.
//
// - monitor: An instance of DropTargetMonitor.
//   You use it to connect state from the React DnD to props.
//   Available functions to get state include canDrop(), isOver() and didDrop()

let collect = (connect, monitor) => {
  return {
    connectDropTarget: connect.dropTarget(),
    isOver: monitor.isOver(),
    canDrop: monitor.canDrop()
  };
};

```

注意这里创建的 `props` 属性的名称恰好和来自 `connect` 及 `monitor` 对象的方法相同或相似，不过其实你可以使用任何名称(如 `draggingSomethingOverMe: monitor.isOver()`)。

所有这些 `props` 属性都会 `render` 函数中用到。`connectDropTarget` 属性应该返回：这个组件中的哪一部分 DOM 是能接收可拖曳对象的目标区域。为简化起见，你可以把整个 `div` 对象作为放置目标。

`isOver` 和 `canDrop` 属性用于在用户将一个元素拖曳到购物车上方时，为购物车显示不同的文字和背景颜色。更新后的 `render` 函数如下所示：

```

render() {
  const { canDrop, isOver, connectDropTarget } = this.props;
  const isActive = canDrop && isOver;

  let backgroundColor = '#FFFFFF';
  if (isActive) {
    backgroundColor = '#F7F7BD';
  } else if (canDrop) {

```

```

    backgroundColor = '#F7F7F7';
  }

  const style = {
    backgroundColor: backgroundColor
  };

  return connectDropTarget(
    <div className='shopping-cart' style={style}>
      {isActive ?
        'Hummmm, snack!' :
        'Drag here to order!'
      }
    </div>
  );
}

```

在更新后的 render 方法中，有一些需要注意的事项：

- 我们使用了解构赋值的形式声明了 canDrop、isOver 和 connectDropTarget 属性的快捷方式(所以之后在使用过程中可以直接写 canDrop，而不需要 this.props.canDrop)。
- 根据用户是否正在拖曳内容、拖曳内容是否在购物车上，组件的背景颜色会有所不同。
- 文字同样也不一样：默认情况下它会显示“Drag here to order”，而当用户拖曳了一个条目到购物车上方时，它将显示“Hummmm, snack!”。
- 和之前直接返回 div 不同，现在使用 connectDropTarget 对 div 进行了封装。

最后我们需要做的事情，就是使用 DropTarget 导出封装之后的高阶组件。此外，因为我们对组件的 props 属性进行了注入，我们也借此机会声明一下 propTypes。代码清单 4-9 展示了 ShoppingCart 组件的完整源代码。

代码清单 4-9: ShoppingCart 高阶组件的完整源代码

```

import React, { PropTypes, Component } from 'react';
import { DropTarget } from 'react-dnd';

// ShoppingCart DND Spec
// "A plain object implementing the drop target specification"
//
// - DropTarget Methods (All optional)
// - drop: Called when a compatible item is dropped.
// - hover: Called when an item is hovered over the component.
// - canDrop: Use it to specify whether the drop target is able to accept
//           the item.
const ShoppingCartSpec = {
  drop() {
    return { name: 'ShoppingCart' };
  }
}

```

```

}
};

// ShoppingCartDropTarget - collect
// "The collecting function."
//
// - connect: An instance of DropTargetConnector.
//   You use it to assign the drop target role to a DOM node.
//
// - monitor: An instance of DropTargetMonitor.
//   You use it to connect state from the React DnD to props.
//   Available functions to get state include canDrop(), isOver() and didDrop()
let collect = (connect, monitor) => {
  return {
    connectDropTarget: connect.dropTarget(),
    isOver: monitor.isOver(),
    canDrop: monitor.canDrop()
  };
};

```

```

class ShoppingCart extends Component {
  render() {
    const { canDrop, isOver, connectDropTarget } = this.props;
    const isActive = canDrop && isOver;

```

```

    let backgroundColor = '#FFFFFF';

```

```

    if (isActive) {
      backgroundColor = '#F7F7BD';
    } else if (canDrop) {
      backgroundColor = '#F7F7F7';
    }

```

```

    const style = {
      backgroundColor: backgroundColor
    };

```

```

    return connectDropTarget(
      <div className='shopping-cart' style={style}>
        {isActive ?
          'Hummmm, snack!' :
          'Drag here to order!'
        }
      </div>
    );

```

```

ShoppingCart.propTypes = {
  connectDropTarget: PropTypes.func.isRequired,

```

```

    isOver: PropTypes.bool.isRequired,
    canDrop: PropTypes.bool.isRequired
  }

```

```

export default DropTarget("snack", ShoppingCartSpec, collect)
  (ShoppingCart);

```

注意 `DropTarget` 高阶封装器的 `type` 参数中，我们使用了可放置到该组件中的拖曳源的 `type` (在这个示例中，就是 `'snack'`)。

#### 4. Snack 组件

接下来，实现 `Snack` 组件，过程和实现 `ShoppingCart` 组件类似。代码清单 4-10 展示了该组件的基础架构：

代码清单 4-10: `Snack` 组件的基本结构

```

import React, { Component, PropTypes } from 'react';
import { DragSource } from 'react-dnd';

class Snack extends Component {
  render() {
    const { name } = this.props;

    const style = {
      opacity: 1
    };

    return (
      <div className='snack' style={style}>
        {name}
      </div>
    )
  }
}

Snack.propTypes = {
  name: PropTypes.string.isRequired
};

```

在这个基础结构中，`Snack` 组件接收一个 `name` 属性并将其渲染在一个 `div` 标签中。组件中还包含一个内嵌样式，将当前透明度设置为 1。

接下来实现 `spec` 对象。你需要响应 `beginDrag` 和 `endDrag` 事件。在 `beginDrag` 中，返回一个字符串(类似于 `ShoppingCart` 的 `drop` 事件)。在 `endDrag` 中，需要对最终返回的值做一些处理。你会得到正在拖曳的元素返回的字符串、正准备放置的目标元素返回的字符串，然后把它们输出到控制台上。代码清单 4-11 中展示了更新后的 `Snack` 组件，其中包含 `spec` 对象。



代码清单 4-11: Snack 组件中 spec 对象的实现

```

import React, { Component, PropTypes } from 'react';
import { DragSource } from 'react-dnd';

// snack Drag'nDrop spec
//
// - Required: beginDrag
// - Optional: endDrag
// - Optional: canDrag
// - Optional: isDragging
const snackSpec = {
  beginDrag(props) {
    return {
      name: props.name
    };
  },
  endDrag(props, monitor) {
    const dragItem = monitor.getItem();
    const dropResult = monitor.getDropResult();

    if (dropResult) {
      console.log(`You dropped ${dragItem.name} into ${dropResult.name}`);
    }
  }
};

class Snack extends Component {
  render() { ... }
}
Snack.propTypes = { ... }

```

作为 Snack 组件的最后一步,接下来实现 collect 函数,这里会连接拖曳元素的 DOM,并把拖曳状态映射为组件的 props 属性。因为我们正在映射拖曳状态,可以借此机会做两件事:声明额外的 propTypes;在内嵌样式规则中使用 isDragging 属性,在元素拖曳过程中改变其透明度。最后,我们使用 dragSource 封装导出高阶组件。代码清单 4-12 展示了完整的源代码。

代码清单 4-12: Snack 组件的完整源代码

```

import React, { Component, PropTypes } from 'react';
import { DragSource } from 'react-dnd';

// snack Drag'nDrop spec
//
// - Required: beginDrag
// - Optional: endDrag
// - Optional: canDrag

```

```

// - Optional: isDragging
const snackSpec = {
  beginDrag(props) {
    return {
      name: props.name
    };
  },
  endDrag(props, monitor) {
    const dragItem = monitor.getItem();
    const dropResult = monitor.getDropResult();

    if (dropResult) {
      console.log(`You dropped ${dragItem.name} into ${dropResult.name}`);
    }
  }
};

// Snack DragSource collect collecting function.
// - connect: An instance of DragSourceConnector.
//       You use it to assign the drag source role to a DOM node.
//
// - monitor: An instance of DragSourceMonitor.
//       You use it to connect state from the React DnD to your component's
properties.
// Available functions to get state include canDrag(), isDragging(),
getItemType(),
// getItem(), didDrop() etc.
let collect = (connect, monitor) => {
  return {
    connectDragSource: connect.dragSource(),
    isDragging: monitor.isDragging()
  };
};

class Snack extends Component {
  render() {
    const { name, isDragging, connectDragSource } = this.props;
    const opacity = isDragging ? 0.4 : 1;

    const style = {
      opacity: opacity
    };

    return (
      connectDragSource(
        <div className='snack' style={style}>
          {name}
        </div>
      )
    )
  }
}

```

```

    });
  }
}

```

```

Snack.propTypes = {
  connectDragSource: PropTypes.func.isRequired,
  isDragging: PropTypes.bool.isRequired,
  name: PropTypes.string.isRequired
};
export default DragSource('snack', snackSpec, collect)(Snack);

```

## 5. 样式

作为结束，你只需要再编写一些样式就可以了，如代码清单 4-13 所示。

代码清单 4-13: 项目的样式表

```

body {
  font: 16px/1 sans-serif;
}
#root{
  height: 100%;
}
h1 {
  font-weight: 200;
  color: #3b414c;
  font-size: 20px;
}
.app{
  white-space: nowrap;
  height: 100%;
}
.snack{
  display: inline-block;
  padding: .5em;
  margin: 0 1em 1em 0.25em;
  border: 4px solid #d9d9d9;
  background: #f7f7f7;
  height: 5rem;
  width: 5rem;
  border-radius: 5rem;
  cursor: pointer;
  line-height: 5em;
  text-align: center;
  color: #333;
}
.shopping-cart{
  border: 5px dashed #d9d9d9;
  border-radius: 10px;
}

```

```
padding: 5rem 2rem;
text-align: center;
}
```

在浏览器中测试一下，你会发现这个示例代码已经能够正常工作了，可以把小吃拖放到购物车中。和之前的章节一样，这个示例的完整代码可在网站 [www.apress.com](http://www.apress.com) 及本书的 github 页面(<https://github.com/pro-react>)中找到。

## 6. 重构：使用常量

尽管已经能够在浏览器中运行了，在真正完成这个示例之前还有一个必要的调整：`dragSource` 和 `dropTarget` 都需要一个 `type` 参数作为拖曳组件的唯一标识。到目前为止，你只是简单地在 `Snack` 组件和 `ShoppingCart` 组件中使用了同样的字符串("snack")，不过在不同文件中手动地输入这种唯一标识需要确保字符串是一模一样的，这很可能会犯错误。

处理这种情况最好的方式，是创建一个用于定义常量的 JavaScript 文件，只读的常量值可以在应用程序的任何地方进行引用。这不只是 React DnD 中的最佳实践，也适用于所有需要在应用程序中跨组件、跨不同文件使用唯一标识的情况。

所以，下面创建一个 `constants.js` 文件。这是一个 JavaScript 模块，导出一个包含 `SNACK` 常量的对象。代码清单 4-14 中展示了其源代码。

### 代码清单 4-14：常量定义了 JavaScript 文件

```
export default {
  SNACK: 'snack'
};
```

接下来，编辑一下 `Snack` 组件和 `ShoppingCart` 组件，导入该文件并引用这个常量，代替之前硬编码进去的"snack"字符串。代码清单 4-15 中展示了更新后的 `ShoppingCart` 组件。代码清单 4-16 展示了更新后的 `Snack` 组件。

### 代码清单 4-15：更新后的 `ShoppingCart` 组件，使用常量作为放置目标的 `type` 参数

```
import React, { PropTypes, Component } from 'react';
import { DropTarget } from 'react-dnd';
import constants from './constants';

const ShoppingCartSpec = {...};
let collect = (connect, monitor) => {...};
class ShoppingCart extends Component {...};
ShoppingCart.propTypes = {...};

export default DropTarget(constants.SNACK, ShoppingCartSpec, collect)
(ShoppingCart);
```

代码清单 4-16: 更新后的 Snack 组件, 使用常量作为拖曳源的 type 参数

```
import React, { Component, PropTypes } from 'react';
import { DragSource } from 'react-dnd';
import constants from './constants';

const snackSpec = {...};
let collect = (connect, monitor) => {...};
class Snack extends Component {...};
Snack.propTypes = {...};

export default DragSource(constants.SNACK, snackSpec, collect)(Snack);
```

## 4.3 看板应用: 支持动画和拖放

下面回到贯穿本书始终的看板应用的开发上来, 我们需要增加应用的动画和拖放支持。你将添加一个简单的过渡动画用于卡片的开启和关闭, 并支持使用拖曳方式在列表之间移动卡片。

### 4.3.1 卡片切换动画

为实现卡片中详细内容显示/隐藏的切换动画, 需要使用 `ReactCSSTransitionGroup` 插件, 首先在看板项目中安装它: `npm install --save react-addons-css-transition-group`。

接下来, 在卡片组件中, 需要导入 `ReactCSSTransitionGroup`, 并在 `cardDetails` 外面做一个封装。在样式表中, 需要创建一个 CSS 过渡动画来改变 `max-height` 属性。代码清单 4-17 展示了 `Card` 组件(变化的部分高亮表示)。代码清单 4-18 中展示了新增的 CSS 样式。

代码清单 4-17: 在 `Card` 组件的详情中使用 `ReactCSSTransitionGroup`

```
import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked'
import CheckList from './CheckList';

let titlePropType = (props, propName, componentName) => {
  if (props[propName]) {
    let value = props[propName];
    if (typeof value !== 'string' || value.length > 80) {
      return new Error(
        `${propName} in ${componentName} is longer than 80 characters`
      );
    }
  }
}
```

```

class Card extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      showDetails: false
    };
  }

  toggleDetails() {
    this.setState({showDetails: !this.state.showDetails});
  }

  render() {
    let cardDetails;
    if (this.state.showDetails) {
      cardDetails = (
        <div className="card_details">
          <span dangerouslySetInnerHTML={{__html:marked(
            this.props.description)}} />
          <CheckList taskCallbacks={this.props.taskCallbacks}
            tasks={this.props.tasks} cardId={this.props.id} />
        </div>
      );
    }

    let sideColor = {
      position: 'absolute',
      zIndex: -1,
      top: 0,
      bottom: 0,
      left: 0,
      width: 7,
      backgroundColor: this.props.color
    };

    return (
      <div className="card">
        <div style={sideColor}/>
        <div className={
          this.state.showDetails? "card_title card_title--is-open" :
            "card_title"
        } onClick={this.toggleDetails.bind(this)}>
          {this.props.title}
        </div>
        <ReactCSSTransitionGroup transitionName="toggle"
          transitionEnterTimeout={250}

```



```

        transitionLeaveTimeout={250} >
        {cardDetails}
      </ReactCSSTransitionGroup>
    </div>
  );
}

Card.propTypes = {
  id: PropTypes.number,
  title: titlePropType,
  description: PropTypes.string,
  color: PropTypes.string,
  tasks: PropTypes.array,
  taskCallbacks: PropTypes.object,
};

export default Card;

```

代码清单 4-18: 添加的 CSS 样式, 用于 max-height 属性的过渡动画

```

.toggle-enter{
  max-height: 0;
  overflow: hidden;
}

.toggle-enter.toggle-enter-active{
  max-height: 300px;
  overflow: hidden;
  transition: max-height .25s ease-in;
}

.toggle-leave{
  max-height: 300px;
  overflow: hidden;
}

.toggle-leave.toggle-leave-active{
  max-height: 0;
  overflow: hidden;
  transition: max-height .25s ease-out;
}

```

### 4.3.2 卡片的拖曳

最后, 我们要实现卡片的拖放功能, 不过和目前为止所做的内容不太一样。这次我们还要把卡片做成可排序的, 所以 you 不仅可在列表之间拖动卡片, 还可在同一个列表中交换它和其他卡片的顺序。首先, 你需要安装 **React DND 2** 和它的 **HTML5** 后端:

```
npm install --save react-dnd@2.x.x react-dnd-html5-backend@1.x.x
```

接下来, 在 `KanbanAppContainer` 组件内部创建两个新的方法, 一个用于更新 `Card` 组件的状态(卡片所在的列表), 另一个用来更新 `Card` 组件的位置。这两个方法和之前实现任务功能的方式(方法、回调函数)相似: 接收 `Card Id`; 找到 `Card` 的索引; 使用静态的帮助方法更新状态信息; 最后设置 `state`(目前为止还没有把 `state` 持久化到服务器上)。代码清单 4-19 展示了更新后的 `KanbanBoardContainer`。

代码清单 4-19: 更新后的 `KanbanBoardContainer` 组件, 添加了 `updateCardStatus` 和 `updateCardPosition` 方法

```
import React, { Component } from 'react';
import KanbanBoard from './KanbanBoard';
import update from 'react-addons-update';
// Polyfills
import 'whatwg-fetch';
import 'babel-polyfill';
const API_URL...
const API_HEADERS...

class KanbanBoardContainer extends Component {
  constructor() {...}
  componentDidMount() {...}

  addTask(cardId, taskName){...}
  deleteTask(cardId, taskId, taskIndex){...}
  toggleTask(cardId, taskId, taskIndex){...}

  updateCardStatus(cardId, listId){
    // Find the index of the card
    let cardIndex = this.state.cards.findIndex((card) => card.id == cardId);
    // Get the current card
    let card = this.state.cards[cardIndex]
    // Only proceed if hovering over a different list
    if(card.status !== listId){
      // set the component state to the mutated object
      this.setState(update(this.state, {
        cards: {
          [cardIndex]: {
            status: { $set: listId }
          }
        }
      })));
    }
  }

  updateCardPosition(cardId, afterId) {
    // Only proceed if hovering over a different card
```

```

if(cardId !== afterId) {
  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((card) => card.id === cardId);
  // Get the current card
  let card = this.state.cards[cardIndex];
  // Find the index of the card the user is hovering over
  let afterIndex = this.state.cards.findIndex((card) => card.id === afterId);
  // Use splice to remove the card and reinsert it at the new index
  this.setState(update(this.state, {
    cards: {
      $splice: [
        [cardIndex, 1],
        [afterIndex, 0, card]
      ]
    }
  })));
}
}

render() {
  return (
    <KanbanBoard cards={this.state.cards}
      taskCallbacks={{
        toggle: this.toggleTask.bind(this),
        delete: this.deleteTask.bind(this),
        add: this.addTask.bind(this)
      }}
      cardCallbacks={{
        updateStatus: this.updateCardStatus.bind(this),
        updatePosition: this.updateCardPosition.bind(this)
      }}
    />
  )
}
}

export default KanbanBoardContainer;

```

需要注意代码中的cardCallbacks对象，其中包含了对新方法的引用，它被传递到KanbanBoard组件中。新的cardCallbacks函数会同时被列表组件(当你拖曳卡片到其他列表中时)和Card组件自身(当你在列表中排序时)调用，所以你必须要在层级结构中对所有组件进行编辑，接收并传递这个属性。这些组件包括KanbanBoard和List。代码清单 4-20 和代码清单 4-21 显示了这两个组件更新后的代码。

代码清单 4-20: Kanban 组件，接收 cardCallbacks props 并将其传递给 List 组件

```

class KanbanBoard extends Component {
  render() {
    return (
      <div className="app">

```

```

    <List id='todo' title="To Do"
      taskCallbacks={this.props.taskCallbacks}
      cardCallbacks={this.props.cardCallbacks}
      cards={ this.props.cards.filter((card) =>
        card.status === "todo") }
    />

    <List id='in-progress' title="In Progress"
      taskCallbacks={this.props.taskCallbacks}
      cardCallbacks={this.props.cardCallbacks}
      cards={ this.props.cards.filter((card) =>
        card.status === "in-progress") }
    />

    <List id='done' title='Done'
      taskCallbacks={this.props.taskCallbacks}
      cardCallbacks={this.props.cardCallbacks}
      cards={ this.props.cards.filter((card) =>
        card.status === "done") }
    />
  </div>
);
}
}
KanbanBoard.propTypes = {
  cards: PropTypes.arrayOf(PropTypes.object),
  taskCallbacks: PropTypes.object,
  cardCallbacks: PropTypes.object
};

```

**export default** KanbanBoard;

代码清单 4-21: List 组件, 接收 cardCallbacks props 并将其传递给 Card 组件

```

class List extends Component {
  render() {
    let cards = this.props.cards.map((card) => {
      return <Card key={card.id}
        taskCallbacks={this.props.taskCallbacks}
        cardCallbacks={this.props.cardCallbacks} {...card} />
    });

    return (...);
  }
}
List.propTypes = {
  id: PropTypes.string.isRequired,
  title: PropTypes.string.isRequired,

```

```

cards: PropTypes.arrayOf(React.PropTypes.object),
taskCallbacks: PropTypes.object,
cardCallbacks: PropTypes.object
};

```

```
export default List;
```

在实现拖曳的准备工作最后，下面创建一个 `constants.js` 文件，声明 CARD 类型，如代码清单 4-22 所示。

代码清单 4-22：常量文件，包含 CARD 类型的定义

```

export default {
  CARD: 'card'
};

```

### 1. 跨列表拖曳

现在你需要使用 React DnD 的高阶组件来设置拖曳源、放置目标和拖曳上下文。DragSource 其实就是 Card 组件，DropTarget 是 List 组件，而上下文则是 KanbanBoard 组件。

下面从 Card 组件开始，将其设置为 DragSource。这个过程和本章早些时候介绍拖放的示例实现中非常类似，不过在 cardSpec 中你不需要实现 endDrag 方法，因为我们需要在拖放过程中让卡片能在列表之间移动，它可能会位于一个新的列表上方。代码清单 4-23 展示了相关的代码。

代码清单 4-23：作为 DragSource 的 Card 组件，只实现了 spec 对象中必需的 beginDrag

```

import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import CheckList from './CheckList';
import { DragSource } from 'react-dnd';
import constants from './constants';

let titlePropType = (props, propName, componentName) => {...}

const cardDragSpec = {
  beginDrag(props) {
    return {
      id: props.id
    };
  }
};

let collectDrag = (connect, monitor) => {
  return {

```

```

    connectDragSource: connect.dragSource()
  };
}

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}

  render() {
    const { connectDragSource } = this.props;

    let cardDetails;
    if (this.state.showDetails) {...}

    let sideColor = {...}

    return connectDragSource(
      <div className="card">
        <div style={sideColor}/>
        <div className={...} onClick={this.toggleDetails.bind(this)}>
          {this.props.title}
        </div>
        <ReactCSSTransitionGroup transitionName="toggle">
          {cardDetails}
        </ ReactCSSTransitionGroup>
      </div>
    );
  }
}

Card.propTypes = {
  id: PropTypes.number,
  title: titlePropType,
  description: PropTypes.string,
  color: PropTypes.string,
  tasks: PropTypes.array,
  taskCallbacks: PropTypes.object,
  cardCallbacks: PropTypes.object,
  connectDragSource: PropTypes.func.isRequired
};
export default DragSource(constants.CARD, cardDragSpec, collectDrag)(Card);

```

接下来将 List 组件变成 DropTarget。在它的 spec 对象中，你需要使用一个 hover 方法，在卡片位于列表上方时调用卡片的回调函数来立即更新其状态，这样用户就能立刻看到这个反馈。代码清单 4-24 展示了实现代码。

#### 代码清单 4-24：作为 DropTarget 的列表组件

```

import React, { Component, PropTypes } from 'react';
import { DropTarget } from 'react-dnd';

```



```

import Card from './Card';
import constants from './constants';

const listTargetSpec = {
  hover(props, monitor) {
    const draggedId = monitor.getItem().id;
    props.cardCallbacks.updateStatus(draggedId, props.id)
  }
};

function collect(connect, monitor) {
  return {
    connectDropTarget: connect.dropTarget()
  };
}

class List extends Component {
  render() {
    const { connectDropTarget } = this.props;

    let cards = this.props.cards.map((card) => {
      return <Card key={card.id} taskCallbacks={this.props.taskCallbacks}
        cardCallbacks={this.props.cardCallbacks} {...card} />
    });
    return connectDropTarget(
      <div className="list">
        <h1>{this.props.title}</h1>
        {cards}
      </div>
    );
  }
}

List.propTypes = {
  id: PropTypes.string.isRequired,
  title: PropTypes.string.isRequired,
  cards: PropTypes.arrayOf(React.PropTypes.object),
  taskCallbacks: PropTypes.object,
  cardCallbacks: PropTypes.object,
  connectDropTarget: PropTypes.func.isRequired
}

export default DropTarget(constants.CARD, listTargetSpec, collect)(List);

```

拼图的最后一块，是找到一个 Card 和 List 公共的父级组件，将其作为拖放上下文。这里使用 KanbanBoard 作为这个组件，如代码清单 4-25 所示。

#### 代码清单 4-25：将 KanbanBoard 组件作为拖放上下文

```

import React, { Component, PropTypes } from 'react';
import { DragDropContext } from 'react-dnd';

```

```
import HTML5Backend from 'react-dnd-html5-backend';
import List from './List';

class KanbanBoard extends Component {
  render() {
    return (...);
  }
}
KanbanBoard.propTypes = {...};

export default DragDropContext(HTML5Backend)(KanbanBoard);
```

如果现在进行测试，可在列表之间进行卡片的拖放，然后它会立即更新。下面接着实现排序操作。

## 2. 卡片排序

使用 React DnD 实现条目排序的关键在于：同时把元素作为 `DragSource` 和 `DropTarget`。于是当用户开始拖曳一个元素时，你可以使用 `hover` 处理程序来检测它位于哪个其他元素之上，然后利用这个信息来改变其位置。

`Card` 组件现在已经是一个 `DragSource` 了。下面接着把它变成一个 `DropTarget`：添加一个不同的 `spec` 属性和 `collect` 函数，让其也可以表现为 `DropTarget`。在 `Card` 组件的 `dropSpec` 中，使用 `hover` 函数（就像在 `List` 组件中一样）检测是否有其他的卡片位于它上方。这种情况下，调用 `updatePosition` 回调函数来交换这两个卡片的位置。最后，同样使用 `DropTarget` 高阶组件来将 `Card` 组件导出为 `DropTarget` 组件。代码清单 4-26 中展示了更新后的 `Card` 组件。注意添加的 `cardDropSpec`、`collectDrop` 函数，调用 `connectDropTarget` 并将其导出为 `DropTarget` 的高阶函数。

代码清单 4-26：作为 `DropTarget` 的 `Card` 组件

```
import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from './constants';
import CheckList from './CheckList';

let titlePropType = (props, propName, componentName) => {...};

const cardDragSpec = {
  beginDrag(props) {
    return {
      id: props.id
    };
  },
};

export default connectDropTarget(
  DragDropContext(HTML5Backend)(
    DropTarget(
      DragSource(
        Card,
        cardDragSpec,
        collectDrop
      ),
      cardDropSpec,
      collectDrop
    )
  )
);
```

```

const cardDropSpec = {
  hover(props, monitor) {
    const draggedId = monitor.getItem().id;
    props.cardCallbacks.updatePosition(draggedId, props.id);
  }
}

let collectDrag = (connect, monitor) => {
  return {
    connectDragSource: connect.dragSource()
  };
}

let collectDrop = (connect, monitor) => {
  return {
    connectDropTarget: connect.dropTarget(),
  };
}

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}

  render() {
    const { connectDragSource, connectDropTarget } = this.props;

    let cardDetails;
    if (this.state.showDetails) {...}

    let sideColor = {...}

    return connectDropTarget(connectDragSource(
      <div className='card'>
        <div style={sideColor}/>
        <div className="card__edit"></div>
        <div className={...} onClick={this.toggleDetails.bind(this)}>
          {this.props.title}
        </div>
        <ReactCSSTransitionGroup transitionName="toggle">
          {cardDetails}
        </ ReactCSSTransitionGroup>
      </div>
    ));
  }
}

Card.propTypes = {
  id: PropTypes.number,
  title: titlePropType,
}

```

```

    description: PropTypes.string,
    color: PropTypes.string,
    tasks: PropTypes.array,
    taskCallbacks: PropTypes.object,
    cardCallbacks: PropTypes.object,
    connectDragSource: PropTypes.func.isRequired,
    connectDropTarget: PropTypes.func.isRequired
  };

  const dragHighOrderCard = DragSource(constants.CARD, cardDragSpec,
    collectDrag)(Card);
  const dragDropHighOrderCard = DropTarget(constants.CARD, cardDropSpec,
    collectDrop)(dragHighOrderCard);
  export default dragDropHighOrderCard

```

值得庆祝！现在你既可以在列表之间拖曳一个卡片，也可以在列表内的其他卡片之间来回移动了。唯一没做的就是把变化持久化到服务器中。如果你移动卡片位置之后刷新浏览器，它还是会回到原始的位置。

### 3. 节流回调函数

在用户拖曳一个卡片时，会触发大量的回调函数。将卡片成功移动到其他卡片上方时会调用 `updatePosition` 回调函数，而移动到不同的列表时则会调用 `updateStatus` 回调函数。

像这样每秒内调用 `card` 中的回调函数好几十次，会给拖放操作造成潜在的性能问题，因此我们需要实现一个节流函数。节流函数会接收两个参数：需要节流的原始函数 `func` 以及 `wait`。它会返回参数函数的节流版本，当反复调用该函数时，在每个由 `wait` 参数指定的毫秒数之内，最多只会执行一次原始函数。我们实现的节流函数会足够智能，在调用参数发生变化时，会立即执行。

为保持项目有组织性，需要在一个新的 JavaScript 文件 `utils.js` 中创建这个节流函数，如代码清单 4-27 所示。

代码清单 4-27: `utils.js` JavaScript 模块中的节流函数

```

export const throttle = (func, wait) => {
  let context, args, prevArgs, argsChanged, result;
  let previous = 0;
  return function() {
    let now, remaining;
    if (wait) {
      now = Date.now();
      remaining = wait - (now - previous);
    }

    context = this;
    args = arguments;
    argsChanged = JSON.stringify(args) !== JSON.stringify(prevArgs);
    prevArgs = {...args};
    if (argsChanged || wait && (remaining <= 0 || remaining > wait)) {

```

```

    if(wait){
      previous = now;
    }
    result =func.apply(context, args);
    context = args =null;
  }
  return result;
};
};

```

接下来，编辑一下 KanbanBoardContainer，创建节流版本的 updateCardPosition 和 updateCardStatus。首先，导入节流工具方法，然后在 KanbanBoardContainer 的构造函数中创建节流版本的 updateCardPosition 和 updateCardStatus。最后，更新 render 方法中的 cardCallback 对象，将节流版本的方法传给 Kanban 组件。代码清单 4-28 中展示了更新后的 KanbanBoardContainer 源代码。

代码清单 4-28：节流版本的 updateCardStatus 和 updateCardPosition 方法

```

import React, { Component } from 'react';
import update from 'react-addons-update';
import {throttle} from './utils';
import KanbanBoard from './KanbanBoard';
// Polyfills
import 'whatwg-fetch';
import 'babel-polyfill';

const API_URL = '...';
const API_HEADERS = {...};

class KanbanBoardContainer extends Component {
  constructor(){
    super(...arguments);
    this.state = {
      cards:[],
    };
    // Only call updateCardStatus when arguments change
    this.updateCardStatus = throttle(this.updateCardStatus.bind(this));
    // Call updateCardPosition at max every 500ms (or when arguments change)
    this.updateCardPosition =
      throttle(this.updateCardPosition.bind(this),500);
  }

  componentDidMount(){...}
  addTask(cardId, taskName){...}
  deleteTask(cardId, taskId, taskIndex){...}
  toggleTask(cardId, taskId, taskIndex){...}
  updateCardStatus(cardId, listId) {...}
  updateCardPosition(cardId , afterId){...}

```

```

render() { return (
  <KanbanBoard cards={this.state.cards}
    taskCallbacks={{
      toggle: this.toggleTask.bind(this),
      delete: this.deleteTask.bind(this),
      add: this.addTask.bind(this) }}
    cardCallbacks={{
      updateStatus: this.updateCardStatus,
      updatePosition: this.updateCardPosition
    }} />
)
}
}

export default KanbanBoardContainer;

```

此时再做测试的话，一切应该都和之前一样，只是解决了潜在可能发生的性能问题。在下一个主题中，我们将把卡片的状态持久化到服务器中。

#### 4. 持久化新的卡片位置和状态

关于持久化Card组件新state的功能，我们可能首先想到的就是在KanbanBoardContainer组件的updateCardStatus和updateCardPosition方法内实现该功能。但问题在于这个时候用户依然在拖曳卡片的过程中，在确定最终位置之前，它可能会移动到很多不同的卡片上或者列表中。如果你在这两个方法内进行持久化，就意味着在这个过程中需要反复调用服务器端，这不仅有显而易见的性能问题，在服务器出错时想要回滚也会变得非常困难。

取而代之的做法是，在用户开始拖曳一个卡片时，注册卡片的id和状态，然后在拖曳结束时再调用服务器端。如果操作失败，还可以根据之前保存的内容恢复卡片的状态。

为了实现这个功能，需要在KanbanBoardContainer容器中创建一个名为persistCardDrag的新方法。在这个新方法中，使用fetch函数调用Kanban API，传递卡片的新状态和位置。如果调用失败，需要在界面中把卡片恢复到之前的状态。此外，还需要让persistCardDrag方法在cardCallbacks对象内部可见(从而可在Card组件中调用)。代码清单4-29显示了更新后的KanbanBoardContainer。

代码清单 4-29: 带有 persistCardDrag 方法的 KanbanBoardContainer

```

import React, { Component } from 'react';
import update from 'react-addons-update';
import {throttle} from './utils';
import KanbanBoard from './KanbanBoard';
// Polyfills
import 'whatwg-fetch';
import 'babel-polyfill';

const API_URL...

```



```

const API_HEADERS...

class KanbanBoardContainer extends Component {
  constructor(){...}

  componentDidMount(){...}

  addTask(cardId, taskName){...}
  deleteTask(cardId, taskId, taskIndex){...}
  toggleTask(cardId, taskId, taskIndex){...}
  updateCardPosition (cardId , afterId) {...}
  updateCardStatus(cardId, listId){...}

  persistCardDrag (cardId, status) {
    // Find the index of the card
    let cardIndex = this.state.cards.findIndex((card)=>card.id === cardId);
    // Get the current card
    let card = this.state.cards[cardIndex]

    fetch(`${API_URL}/cards/${cardId}`, {
      method: 'put',
      headers: API_HEADERS,
      body: JSON.stringify({status: card.status, row_order_position:
        cardIndex})
    })
    .then((response) => {
      if(!response.ok){
        // Throw an error if server response wasn't 'ok'
        // so you can revert back the optimistic changes
        // made to the UI.
        throw new Error("Server response wasn't OK")
      }
    })
    .catch((error) => {
      console.error("Fetch error:",error);
      this.setState(
        update(this.state, {
          cards: {
            [cardIndex]: {
              status: { $set: status }
            }
          }
        })
      );
    });
  }

  render() {

```

```

    return (
      <KanbanBoard cards={this.state.cards}
        taskCallbacks={{
          toggle: this.toggleTask.bind(this),
          delete: this.deleteTask.bind(this),
          add: this.addTask.bind(this)
        }}
        cardCallbacks={{
          updateStatus: this.updateCardStatus,
          updatePosition: this.updateCardPosition,
          persistCardDrag: this.persistCardDrag.bind(this)
        }}
      />
    )
  }
}

export default KanbanBoardContainer;

```

接下来，只需要在用户结束拖曳时，在 Card 组件的 cardDragSpec 中调用 persistDrag 方法。代码清单 4-30 展示了更新后的 Card 组件。

#### 代码清单 4-30：更新后的 Card 组件，调用 persistCardDrag 方法

```

import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from './constants';
import CheckList from './CheckList';

let titlePropType = (props, propName, componentName) => {...}

const cardDragSpec = {
  beginDrag(props) {
    return {
      id: props.id,
      status: props.status
    };
  },
  endDrag(props) {
    props.cardCallbacks.persistCardDrag(props.id, props.status);
  }
}

const cardDropSpec = {...}

let collectDrag = (connect, monitor) => {...}

```

```

let collectDrop = (connect, monitor) => {...}

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}
  render() {...}
}

Card.propTypes = {...}

let dragHighOrderCard = DragSource(constants.CARD, cardDragSpec,
  collectDrag)(Card);
let dragDropHighOrderCard = DropTarget(constants.CARD, cardDropSpec,
  collectDrop)
  (dragHighOrderCard);
export default dragDropHighOrderCard

```

## 4.4 本章小结

在本章中，你了解了如何使用 CSS 动画(通过 React 的 `CSSTransitionGroup` 插件)来实现一个现代、流畅的复杂用户界面，以及如何通过外部的 React DnD 库实现拖放功能。

URL 对于 Web 应用程序而言是一项强大的优势，使得 Web 应用程序比原生应用程序更好。URL 最开始是一个指向一台服务器上某个文档的指针，但是在一个 Web 应用程序中，看待 URL 的最佳方式，应当是将其看成应用程序当前状态的一种表示。通过查看 URL，用户就能知道他现在位于应用程序的哪个位置，用户可将 URL 复制下来等到以后再直接使用它，或是将 URL 传给别人(或其他地方)。

## 5.1 使用原生方式实现路由

为了解基本的路由功能是如何运作的，以及在比基本的无嵌套导航场景稍复杂的情况下会引发的复杂性，下面先实现一个简单的组件，它依赖当前的 URL，来渲染不同的子组件。你将创建一个使用 GitHub API 的应用程序，来返回一个给本书用户使用的资料库列表。除了这个“资料库”部分外，应用程序还有一个首页和一个“关于(About)”部分。下面关注在主组件和路由代码，如代码清单 5-1 所示。

代码清单 5-1：一个基于 URL 渲染不同子组件的组件

```
import React, { Component } from 'react';
import { render } from 'react-dom';

import About from './About';
import Home from './Home';
import Repos from './Repos';

class App extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      route: window.location.hash.substr(1)
    };
  }

  componentDidMount() {
    window.addEventListener('hashchange', () => {
      this.setState({
```

```

        route: window.location.hash.substr(1)
      });
    });
  }

  render() {...}
}

render(<App />, document.getElementById('root'));

```

上面的代码颇为直观。在组件的构造函数中，我们获取 URL 路径的 hash 值，并将它赋给 state 中的 route。为简单起见，现在你还不需要去使用 HTML5 URL History API。然后，当组件装载时，你添加一个事件侦听程序，每当 URL 更改时，就会更新 state 中的 route，组件也将重新渲染。说到渲染，需要做的是在 render 方法中，根据当前的 route 值来使用适当的组件，如代码清单 5-2 所示。

代码清单 5-2: render 方法中根据当前 state 中的 route 值来渲染不同的组件

```

render() {
  var Child;
  switch (this.state.route) {
    case '/about': Child = About; break;
    case '/repos': Child = Repos; break;
    default:       Child = Home;
  }

  return (
    <div>
      <header>App</header>
      <menu>
        <ul>
          <li><a href="#/about">About</a></li>
          <li><a href="#/repos">Repos</a></li>
        </ul>
      </menu>
      <Child/>
    </div>
  )
}

```

在上面的这里示例中，所有表示内部导航页的子组件的结构都是相同的(但是有各自不同的标题)，如代码清单 5-3 到代码清单 5-5 所示。

代码清单 5-3: Home 组件

```

import React, { Component } from 'react';

class Home extends Component {

```

```

render() {
  return (
    <h1>HOME</h1>
  );
}
}

export default Home;

```

#### 代码清单 5-4: About 组件

```

import React, { Component } from 'react';

class About extends Component {
  render() {
    return (
      <h1>ABOUT</h1>
    );
  }
}

export default About;

```

#### 代码清单 5-5: Repos 组件

```

import React, { Component } from 'react';

class Repos extends Component {
  render() {
    return (
      <h1>Github Repos</h1>
    );
  }
}

export default Repos;

```

现在路由系统就已经开始工作了,如果你给应用程序添加上一些样式,它看起来就像是图 5-1 的样子。在这个示例中使用的示范 CSS 如代码清单 5-6 所示。

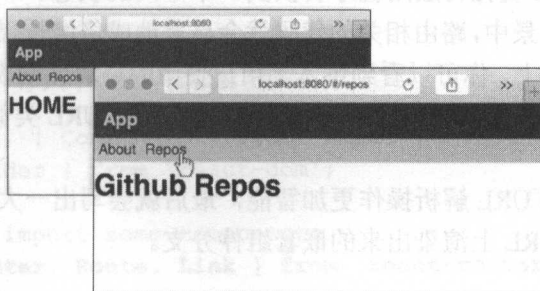


图 5-1 使用路由的示例



## 代码清单 5-6: 路由示例应用程序的 CSS

```

body {
  margin: 0;
  font: 16px/1 sans-serif;
}
menu ul{
  margin: 0;
  padding: 0;
}
menu li {
  display: inline-block;
  padding: 5px;
}
a.active {
  color: #444;
  font-weight: bold;
  text-decoration: none;
}
header {
  padding: 10px;
  background-color: #333;
  color: #ccc;
  font-size: 20px;
  font-weight: bold;
}
menu {
  background-color: #ccc;
  padding: 5px;
  margin-top: 0;
  margin-bottom: 10px;
}

```

虽然这个示例应用程序中的路由系统工作得还不错,但它至少有两个问题,一个问题偏向于概念方面,另一个问题则偏向于实践方面:

- 在这个示例实现中,URL 的维护在整个应用程序运转中具有中心地位:应用程序并不能随着程序状态的转换来自动更新 URL,相反,你是在侦听 URL 的变化,然后随着 URL 变化将应用程序转换到一个不同的状态。
- 在更复杂的场景中,路由相关的代码就会显著地成倍增长。想象这样的一个场景:在 Repos 页面上,你可以看到供本书用户使用的资料库列表,列表中的每一项都可以单击,每次单击都需要不同的路由代码处理,URL 类似于 `/repos/repo_id`(如图 5-2 所示)。
- 你必须让你的 URL 解析操作更加智能,最后就会写出一大堆代码,来处理需要在任何给定 URL 上渲染出来的嵌套组件分支。



图 5-2 展示嵌套的路由

为能处理比基本单层路由更复杂的场景，推荐的方法是使用 React Router 库。React Router 库的声明式 API 能内置地处理嵌套的 URL 和嵌套的组件层级，这个库尽管不在 React 核心库中，但是 React 社区还是像对待 React 标准库那样推崇它。

## 5.2 React Router

React Router 是为一个 React 应用程序添加路由功能的最流行的解决方案。通过将组件与路由(在任何一个嵌套层级)关联起来，它使 UI 能与 URL 保持同步。当用户更改 URL 时，组件就被自动卸载并加载。React Router 库的另一个优势是，它提供了一种机制来让你可以控制应用程序的流程，不管用户是通过编程方式进入某个状态，还是用户通过单击一个 URL 来到达某个状态，代码都可以在不同情况下以一致的方式来运行。

由于 React Router 是一个外部库，所以它必须通过 npm 来进行安装(同时还需要安装 History 库，因为 React Router 依赖于它)。要安装这两个库的第 1 版，使用如下命令：npm install --save react-router@1.x.x history@1.x.x。

React Router 提供了如下 3 个组件：

- Router 和 Route：用来采用声明方式将路由映射到你的应用程序的 UI 层级。
- Link：用来使用恰当的 href 来创建一个完全可访问的锚定标记。当然这不是唯一一种在项目中导航的方法，但是这通常是用户最主要使用的一种方式。

下面修改上节中的那个示例，不再使用“原生”的路由实现，而是改用 React Router。安装了 React Router 库后，首先需要在你的应用程序组件中正确导入它，如代码清单 5-7 所示。

### 代码清单 5-7：导入 React Router 组件

```
import React, { Component } from 'react';
import { render } from 'react-dom';

// first we import some components
import { Router, Route, Link } from 'react-router';
```

```
import About from './About';
import Home from './Home';
import Repos from './Repos';

class App extends Component {...}
```

在 App 类里面，你不再需要构造函数和 `componentDidMount` 方法来管理 URL 解析和注册事件侦听程序；现在这些事情都会被自动处理。在 `render` 方法中，你可以不再使用 `switch` 语句；React Router 会自动根据当前的路由，将 `children` 这个 `props` 设置为相应的组件。同时注意，你需要将 `<a>` 标签替换为 `<Link>` 组件，来生成适当的导航链接。代码清单 5-8 显示了更新后的 App 组件的类。

#### 代码清单 5-8：更新后的 App 组件类

```
class App extends Component {
  render() {
    return (
      <div>
        <header>App</header>
        <menu>
          <ul>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/repos">Repos</Link></li>
          </ul>
        </menu>
        {this.props.children}
      </div>
    );
  }
}
```

最后，你需要声明你的路由。声明的位置是在代码文件的底部。之前我们是将 App 组件渲染到 DOM 上，但现在改成将带有路由信息的 Router 组件传递给 React DOM `render` 方法，如代码清单 5-9 所示。

#### 代码清单 5-9：更改后的渲染方法

```
render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About}/>
      <Route path="repos" component={Repos}/>
    </Route>
  </Router>
), document.getElementById('root'));
```

App.js 的完整代码如代码清单 5-10 所示。

## 代码清单 5-10: 使用 React Router 的完整代码

```

import React, { Component } from 'react';
import { render } from 'react-dom';

import { Router, Route, Link } from 'react-router';

import About from './About';
import Repos from './Repos';
import Home from './Home';

class App extends Component {
  render() {
    return (
      <div>
        <header>App</header>
        <menu>
          <ul>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/repos">Repos</Link></li>
          </ul>
        </menu>
        {this.props.children}
      </div>
    );
  }
}

render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About}/>
      <Route path="repos" component={Repos}/>
    </Route>
  </Router>
), document.getElementById('root'));

```

**提示:**

命名组件: 通常一个路由只映射到单个组件, 这样的话, 只需要在父组件上使用 `this.props.children` 就可以了。但是还可以在设置路由时声明一个或多个命名组件。这种情况下, 父组件就需要在 `props.children` 上通过组件的名称来使用组件, 例如:

```

React.render((
  <Router>
    <Route path="/" component={App}>
      <Route path="groups" components={{content: Groups, sidebar:
        GroupsSidebar}}/>
      <Route path="users" components={{content: Users, sidebar:
        UsersSidebar}}/>
    </Route>
  </Router>
), document.getElementById('root'));

```

```
    </Route>
  </Router>
), element);
```

然后，在组件中：

```
render() {
  return (
    <div>
      {this.props.children.sidebar}-{this.props.children.content}
    </div>
  );
}
```

### 5.2.1 Index 路由

如果你现在进行测试，你会看到一切都如期望那样的工作。但是有一点和原来的实现不同。你不再在任何路由中显示 Home 组件了。如果你在 “/” 路由上访问服务器，它会渲染一个不包含任何子组件的 App 组件，如图 5-3 所示。

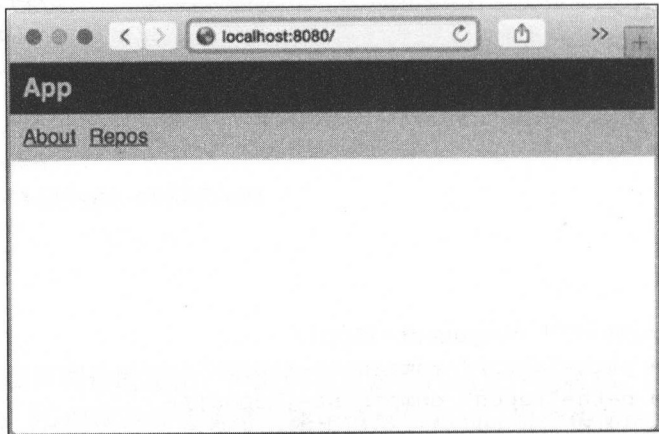


图 5-3 Home 路由渲染了不带任何子组件的 App 组件

你首先想到的是只需要在 router 中添加一个新的 Home 路由，但是你要使用什么路径呢？

```
<Router>
  <Route path="/" component={App}>
    <Route path="???" component={Home}/>
    <Route path="about" component={About}/>
    <Route path="repos" component={Repos}/>
  </Route>
</Router>
```

这里，你可以使用一个<IndexRoute>。只需要导入一个额外的组件，然后使用它来配置 Index 路由即可，如下面的代码和图 5-4 所示。

```

import React, { Component } from 'react';
import { render } from 'react-dom';

import { Router, Route, IndexRoute, Link } from 'react-router';

import About from './About';
import Repos from './Repos';
import Home from './Home';

class App extends Component {...}

render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
      <Route path="about" component={About} />
      <Route path="repos" component={Repos} />
    </Route>
  </Router>
), document.getElementById('root'));

```

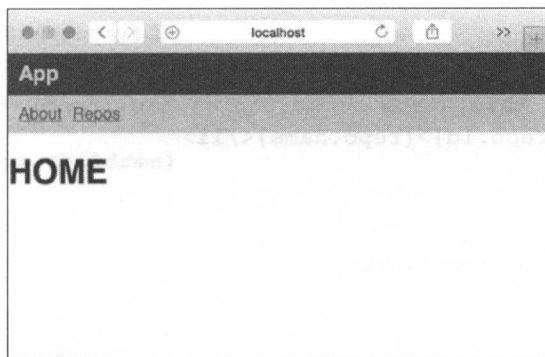


图 5-4 如果没有提供路由，则渲染 Home 组件

### 5.2.2 带参数的路由

现在你已经实现了与之前那个“原生”路由相同的功能，我们想开始在 `Repos` 组件从 GitHub API 实际获取数据。这里需要做的都是已经学过的东西：你将为资料库创建一个本地 `state`，然后在 `componentDidMount` 生命周期方法中访问 API，就和在之前的示例中做的一模一样。代码清单 5-11 显示了增加额外数据访问功能的 `Repos` 组件的代码。

#### 注意：

在这个示例代码中，你将使用本书之前的示例中就用过的新 `window.fetch` 函数。由于旧浏览器并不支持这个新标准，所以你需要确保从 `npm` 安装和请求了 `whatwg-fetch` 模块。

```
npm install --save whatwg-fetch
```



## 代码清单 5-11: 从 GitHub API 获取数据的 Repos 组件

```

import React, { Component } from 'react';

import 'whatwg-fetch';

class Repos extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      repositories: []
    };
  }

  componentDidMount() {
    fetch('https://api.github.com/users/pro-react/repos')
      .then((response) => response.json())
      .then((responseData) => {
        this.setState({repositories: responseData});
      });
  }

  render() {
    let repos = this.state.repositories.map((repo) => (
      <li key={repo.id}>{repo.name}</li>
    ));
    return (
      <div>
        <h1>Github Repos</h1>
        <ul>
          {repos}
        </ul>
      </div>
    );
  }
}

export default Repos;

```

## 注意:

GitHub API 有一个对于未注册用户的限制, 每小时最多只能有 60 次请求。要了解有关 GitHub API 的更新信息, 请访问 <https://developer.github.com/v3/>。

如果你测试应用程序, 当浏览到 Repo 组件时, 你就会看到一个资料库的列表。接下来, 你将创建一个新路由, 通过它来显示详细的资料库信息。这个路由的规则要让 URL 使用 `/repos/details/repo_name` 这样的格式。

你需要创建一个新的 RepoDetails 组件, 然后在 App.js 文件中更新路由。但在做这些之前, 下面修改 Repos 组件, 增加指向资料库列表的链接, 并将 RepoDetails 作为一个

嵌套的子组件加载。更新后的代码如代码清单 5-12 所示(不需要更新的代码已省略)。

代码清单 5-12: 为资料库列表添加 Link 组件, 并渲染嵌套的 RepoDetails 组件

```
import React, { Component } from 'react';
import 'whatwg-fetch';
import { Link } from 'react-router';

class Repos extends Component {
  constructor(...args) {
    super(...args);
  }

  componentDidMount() {
    // ...
  }

  render() {
    let repos = this.state.repositories.map((repo) => (
      <li key={repo.id}>
        <Link to={"/repos/details/"+repo.name}>{repo.name}</Link>
      </li>
    ));
    return (
      <div>
        <h1>Github Repos</h1>
        <ul>
          {repos}
        </ul>
        {this.props.children}
      </div>
    );
  }
}

export default Repos;
```

接下来创建 RepoDetails 组件。在代码中有两点值得注意的地方:

- React Router 会将 repo\_name 参数注入到组件的属性里面。你可以使用这个值来访问 GitHub API, 获取项目的详细信息。
- 本书之前的示例都在 componentDidMount 生命周期方法中获取数据。在 RepoDetails 组件里面, 你需要在一个额外的生命周期方法: componentWillReceiveProps, 实现获取数据的逻辑。这是因为组件被加载之后, 每当用户单击不同的资料库时, 就接收到新参数。这种情况下, componentDidMount 方法并不会在用户每次单击时都重新被调用。但 componentWillReceiveProps 方法则会每次被调用。

代码清单 5-13 显示了 RepoDetails 组件的完整代码。

代码清单 5-13: RepoDetails 组件

```
import React, { Component } from 'react';
import 'whatwg-fetch';
```

```

class RepoDetails extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      repository: {}
    };
  }

  fetchData(repo_name) {
    fetch('https://api.github.com/repos/pro-react/' + repo_name)
      .then((response) => response.json())
      .then((responseData) => {
        this.setState({repository: responseData});
      });
  }

  componentDidMount() {
    // The Router injects the key "repo_name" inside the params prop
    let repo_name = this.props.params.repo_name;
    this.fetchData(repo_name)
  }

  componentWillReceiveProps(nextProps) {
    // The Router injects the key "repo_name" inside the params prop
    let repo_name = nextProps.params.repo_name;
    this.fetchData(repo_name)
  }

  render() {
    let stars = [];
    for (var i = 0; i < this.state.repository.stargazers_count; i++) {
      stars.push('★');
    }
    return (
      <div>
        <h2>{this.state.repository.name}</h2>
        <p>{this.state.repository.description}</p>
        <span>{stars}</span>
      </div>
    );
  }
}

export default RepoDetails;

```

要最终实现嵌套的资料库详情的路由，你需要更新主 App.js 文件。你将导入新组件，并更新 Router 组件，将资料库详情的路由实现为 repos 的子路由，并声明名为 repo\_name 的命名参数，如代码清单 5-14 所示。

## 代码清单 5-14: 更新后的 App.js

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import { Router, Route, IndexRoute, Link } from 'react-router';
import Home from './Home';
import About from './About';
import Repos from './Repos';
import RepoDetails from './RepoDetails';

class App extends Component {...}

render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
      <Route path="about" component={About}/>
      <Route path="repos" component={Repos}>
        /* Add the route, nested where we want the UI to nest */
        <Route path="details/:repo_name" component={RepoDetails} />
      </Route>
    </Route>
  </Router>
), document.getElementById('root'));

```

当在一个路由中声明动态片段时(例如, 上面的代码清单里面所声明的 `repo_name`), React Router 会将 URL 中那个部分中所包含的数据, 都注入到组件 `props` 里面的一个参数特性中。

如果你跟着做了上面的所有示例, 那么现在你应当看到如图 5-5 所示的界面。

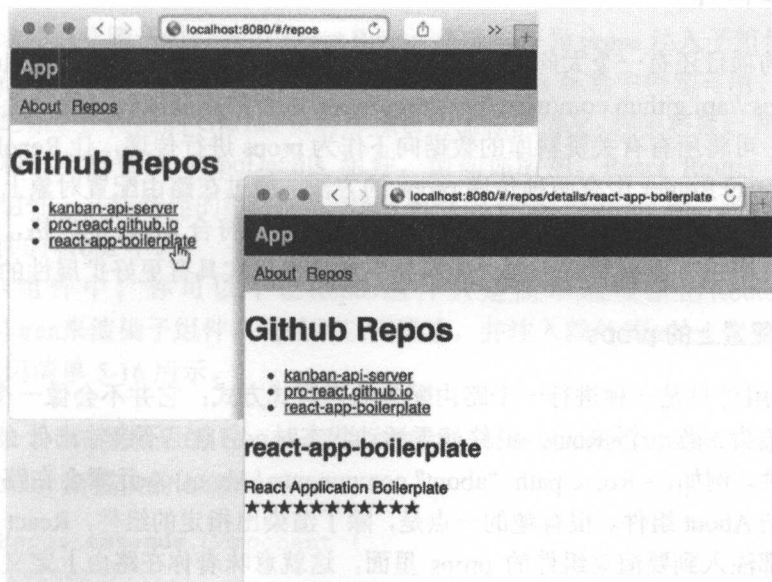


图 5-5 Repos 组件中的嵌套路由

### 5.2.3 设置活动链接

Link 组件有一个额外的优秀功能：它能接受一个名为 `activeClassName` 的可选属性。如果设置了这个属性，Link 组件就会自动将这个 CSS 类应用到活动链接上。下面将这个属性添加到你的 App 组件中，如代码清单 5-15 所示。

代码清单 5-15: App 组件中带有 `activeClassName` 属性的 Link 组件

```
class App extends Component {
  render() {
    return (
      <div>
        <header>App</header>
        <menu>
          <ul>
            <li><Link to="/about" activeClassName=
              "active">About</Link></li>
            <li><Link to="/repos" activeClassName=
              "active">Repos</Link></li>
          </ul>
        </menu>
        {this.props.children}
      </div>
    );
  }
}
```

### 5.2.4 传递 props

现在你的项目还有一个大问题：你正在进行一次多余的数据获取。GitHub API 在你首次访问 <https://api.github.com/users/pro-react/repos> 进行数据获取时，就已经提供了所有资料库详情。可将所有有关资料库的数据向下作为 props 进行传递，让 RepoDetails 组件进行渲染。React Router 中有两种传递 props 的方法：通过在路由配置对象上指定 props，或者在子组件的一个克隆上注入 props。第一个方法要更符合 React 的风格，但它不能解决所有问题。所以下面来看有一点“黑科技”意味但确实具有更好扩展性的做法。

#### 1. 路由配置上的 props

<Route>组件只是一种进行一个路由配置的声明式方式；它并不会像一个常规 React 组件那样被渲染。当一个<Route>组件处于活动状态时，它自己不会输出什么，但会渲染一个指定组件。例如，<Route path="about" component={About} />元素会在路由处于活动状态时，渲染 About 组件。很有趣的一点是，除了渲染出指定的组件，React Router 将所有路由属性都注入到要渲染组件的 props 里面，这就意味着你在路由上定义的所有额外属性，在被渲染的组件中都可以访问到。

为进行演示，下面修改 About 组件，使得它能从路由接收它的 title 属性。首先将一个 title 属性添加到 about 路由上：

```
React.render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
      <Route path="about" component={About} title="About Us" />
      <Route path="repos" component={Repos}>>
        <Route path="details/:repo_name" component={RepoDetails} />
      </Route>
    </Route>
  </Router>
), document.getElementById('root'));
```

接下来，在 About 组件里面，你通过 `this.props.route` 来访问路由配置：

```
import React, { Component } from 'react';
```

```
class About extends Component {
  render() {
    return (
      <h1>{this.props.route.title}</h1>
    );
  }
}
```

```
export default About;
```

## 2. 在子组件上克隆并注入 props

另一个途径是克隆子组件并由 React Router 将数据作为 props 注入子组件，这使你有机会传递正在处理中的额外 props。这个途径在动态 props 场景中特别有用。

这也正是你正在构建的 GitHub 资料库项目所面临的情况。你要将自己在 Repos 组件中获取的资料库数据传给 RepoDetails 组件，但如你所知，React Router 会自动创建 RepoDetails 组件，并将新建的组件注入 Repo 组件的 `props.children` 中，这样你就没机会处理新建组件的 props 了。

在 Repos 组件中，你可以不让 Repos 组件只是简单地根据由 Router 所提供的 `this.props.children` 来渲染子组件，而是可以克隆它，并注入额外的 props (也就是资料库的列表)，如代码清单 5-16 所示。

代码清单 5-16: 更新后的 Repos 组件，它使用 `React.cloneElement` 将额外的 props 传递给由 Router 所提供的 children

```
class Repos extends Component {
  constructor() {...}
  componentDidMount() {...}
```



```

render() {
  let repos = this.state.repositories.map((repo) => (
    <li key={repo.id}>
      <Link to={"/repos/details/"+repo.name}>{repo.name}</Link>
    </li>
  ));

  let child = this.props.children && React.cloneElement(
    this.props.children,
    { repositories: this.state.repositories }
  );

  return (
    <div>
      <h1>Github Repos</h1>
      <ul>
        {repos}
      </ul>
      <div>
        {child}
      </div>
    </div>
  );
}
}

```

现在，RepoDetails组件会被当作一个单纯组件那样对待。它不再有内部的state，它只会接收并显示props。你移除了constructor、componentWillReceiveProps、componentDidMount和fetchData方法，并修改了render方法来基于URL参数查找资料库的数据。代码清单 5-17 显示了更新后的RepoDetails.js文件代码。

### 注意：

之前提示过，Array.prototype.find 是旧浏览器所不支持的新函数。确保你使用 npm install --save babel-polyfill 来安装 Babel，并在 JavaScript 模块中通过 import 'babel-polyfill' 将其导入。

### 代码清单 5-17：更新后的 RepoDetails.js 文件

```

import React, { Component } from 'react';
import 'babel-polyfill';

class RepoDetails extends Component {

  renderRepository() {
    let repository = this.props.repositories.find((repo) => repo.name
      === this.props.params.repo_name);
    let stars = [];
    for (var i = 0; i < repository.stargazers_count; i++) {

```

```

    stars.push('★');
  }
  return (
    <div>
      <h2>{repository.name}</h2>
      <p>{repository.description}</p>
      <span>{stars}</span>
    </div>
  );
}

render() {
  if (this.props.repositories.length > 0) {
    return this.renderRepository();
  } else {
    return <h4>Loading...</h4>;
  }
}
}

export default RepoDetails;

```

## 5.2.5 将 UI 界面与 URL 解耦

虽然现在有关资料库详情的路由已经工作得很好了，但是 URL 片段显得有点长：`/repos/details/:repo_name`。如果你可将 URL 片段改得更短更有针对性一些(比如像这样：`/repo/:repo_name`)，又仍可在 `Repos` 路由下渲染 `RepoDetails` 组件，就更好了。在 `React Router` 中，通过在路由定义中使用一个绝对路径，就可以做到这一点。

所以，你可将之前所使用的：

```

render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
      <Route path="about" component={About} />
      <Route path="repos" component={Repos}>
        <Route path="details/:repo_name" component={RepoDetails} />
      </Route>
    </Route>
  </Router>
), document.getElementById('root'));

```

改为使用绝对路径，如下所示：

```

render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>

```

```

    <Route path="about" component={About} />
    <Route path="repos" component={Repos}>
      <Route path="/repo/:repo_name" component={RepoDetails} />
    </Route>
  </Route>
</Router>
), document.getElementById('root'));
```

当然，你还需要更新 Repo 组件上的链接，来反映出所使用的新 URL，如代码清单 5-18 所示。

代码清单 5-18: Repo 组件上更新后的链接

```

class Repos extends Component {
  constructor() {...}
  componentDidMount() {...}

  render() {
    let repos = this.state.repositories.map((repo) => (
      <li key={repo.id}><Link to={
        "/repo/"+repo.name}>{repo.name}</Link></li>
    ));

    let child = this.props.children && React.cloneElement(...);

    return (...);
  }
}

export default Repos;
```

新的 URL 如图 5-6 所示。

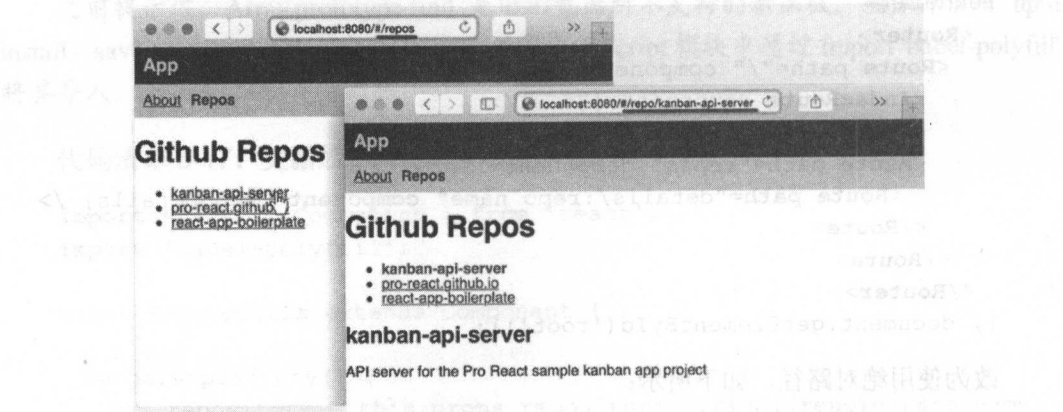


图 5-6 UI 保持了嵌套的层级结构，但使用了自定义解耦的路由

5.2.6 在代码中更改路由

你之前所用的 Link 组件为最终用户提供了一种很好的方式，来在路由间进行转换，但有时你需要在你的组件里，用代码进行路由的转换。你也许想要自动回退，或在某个条件下，将用户重定向到一个不同的路由。

为做到这一点，React Router 将它的 history 对象自动注入它所加载的所有组件中。history 对象负责管理浏览器的访问记录，它提供了表 5-1 所示的用于导航的方法。

表 5-1 history 对象的方法

方法	描述
pushState	转换到一个新 URL 的基本 history 导航方法。你可以选择传递一个参数对象。 例如： <pre>history.pushState(null, '/users/123') history.pushState({showGrades: true}, '/users/123')</pre>
replaceState	与 pushState 的语法相同，但它用一个新的 URL 来替换当前 URL。它有点像一个重定向，但它通过替换 URL，使得 history 的长度保持不变
goBack	回退到导航 history 的上一个位置
goForward	前进到导航 history 中的下一个位置
Go	在导航 history 中前进或后退 n 个位置
createHref	使用路由器的配置来创建一个 URL

为演示 history 的用法，下面创建一个新的 Server Error 路由。如果 fetch 方法无法连接到 API，就从 Repos 组件重定向到这个新路由。代码清单 5-19 和代码清单 5-20 显示了新的 ServerError 组件，以及 App.js 中相应更新后的路由。

代码清单 5-19: 包含一些内置样式的 ServerError 组件

```
import React, { Component } from 'react';  
  
const styles={  
  root:{  
    textAlign:'center'  
  },  
  alert:{  
    fontSize:80,  
    fontWeight: 'bold',  
    color:'#e9ab2d'  
  }  
};  
  
class ServerError extends Component {
```

```

render() {
  return (
    <div style={styles.root}>
      <div style={styles.alert}>&#9888; </div>
      /* &#9888; is the html entity code for the warning character: */
      <h1>Ops, we have a problem</h1>
      <p>Sorry, we could't access the repositories. Please try again in
        a few moments.</p>
    </div>
  );
}
}

export default ServerError;

```

### 代码清单 5-20: App.js 中，更新后的 import 和路由定义

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import { Router, Route, Link, IndexRoute } from 'react-router';

import About from './About';
import Repos from './Repos';
import RepoDetails from './RepoDetails';
import Home from './Home';
import ServerError from './ServerError';

class App extends Component {...}

render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
      <Route path="about" component={About} />
      <Route path="repos" component={Repos}>
        <Route path="/repo/:repo_name" component={RepoDetails} />
      </Route>
      <Route path="error" component={ServerError} />
    </Route>
  </Router>
), document.getElementById('root'));

```

回到 Repos 组件，你在 fetch 的 catch 语句中添加对 pushState 方法的调用，如代码清单 5-21 所示。

### 代码清单 5-21: 更新后的 Repos 组件，使用了 pushState 来重定向到错误页面

```

import React, { Component } from 'react';
import { Link } from 'react-router';
import 'whatwg-fetch';

```





## 5.2.7 History 库

React Router 构建于 History 库之上(还记得当我们使用 npm 安装 React Router 时的情景吗? 我们一并安装了 History 库)。History 库旨在抽象化 URL 和会话管理, 提供一个通用 API, 在不同的浏览器、测试环境和平台上管理导航记录和 URL。

History 库可以有不同的设置。默认情况下, React Router 使用 hash history 设置, 这种设置是使用井号(#, 它的英文名称是 hash)来组成 URL, 创建看起来类似 example.com/#/path 这样的路由。

hash history 是默认设置的原因, 是它可以在旧浏览器上(Internet Explorer 8 和 9)正常工作, 也不需要任何额外的服务器配置。如果你的应用程序不需要兼容旧浏览器, 而且你可对服务器进行配置, 那么更理想的途径是使用浏览器 history 设置, 也就是创建看起来类似 example.com/path 这样的真实 URL。

### 提示:

服务器配置: 浏览器 history 设置可以在不重新加载页面的情况下生成真实的 URL。但是如果用户在一个深层嵌套的 URL 上进行刷新操作, 或者把这个 URL 加入到收藏夹, 会发生什么呢? 这些 URL 都是在浏览器上动态生成的; 它们并非对应于服务器上的真实路径, 因此如果用户首先使用这些 URL 来访问服务器的话, 就很可能会返回一个“页面无法找到”的错误。

要使用浏览器 history 设置, 你需要使用服务器上的 rewrite 配置, 这样当用户在浏览器上直接访问某个路径时, 服务器就会提供首页, 然后 React Router 就会根据 URL 渲染出正确的视图了。

Webpack 开发服务器有一个 historyApiFallback 选项, 可让服务器对于任何未知路径, 都总是渲染首页(如果你正在使用本书的示例应用程序, 那么这个选项已启用)。Node.js 和所有的通用 Web 服务器(例如, Apache 和 Nginx)都有这样的配置。请参考 React Router 的文档和你所使用的服务器的文档。

要实现浏览器 history 设置, 你需要从 History 库导入 createBrowserHistory 方法。然后你就可以调用它, 将生成的浏览器 history 配置作为 Router 组件的 history 属性进行传递。下面在你的示例应用程序中实现这些操作, 如代码清单 5-22 所示。

代码清单 5-22: 更新后的 App.js, 使用了浏览器 history 设置

```
import React, { Component } from 'react';
import { render } from 'react-dom';

import { Router, Route, IndexRoute, Link } from 'react-router';
import createBrowserHistory from 'history/lib/createBrowserHistory';

import About from './About';
import Repos from './Repos';
```

```
import RepoDetails from './RepoDetails';
import Home from './Home';
import ServerError from './ServerError';

class App extends Component {...}

render((
  <Router history={createBrowserHistory()}>
    <Route path="/" component={App}>
      <IndexRoute component={Home}/>
      <Route path="about" component={About} />
      <Route path="repos" component={Repos}>
        <Route path="/repo/:repo_name" component={RepoDetails} />
      </Route>
      <Route path="error" component={ServerError} />
    </Route>
  </Router>
), document.getElementById('root'));
```

## 5.2.8 看板应用：实现路由功能

到现在为止，作为练习示范，看板应用已经起到了它应有的作用，但是它本身作为一个应用还算不上特别有用，因为你还不能编辑或者创建新卡片。下面使用路由来实现这两个功能。`/new` 这个路由会显示一个用来创建新卡片的表单，`/edit/:card_id` 路由则显示一个包含了卡片当前属性，并让用户可以修改的表单。会有两个新组件：`NewCard` 和 `EditCard`，由于它们很多特性都类似(比如要显示出完整的卡片表单)，你还会创建一个 `CardForm` 组件来包含所有要共享的 UI 界面，然后那两个新组件会使用这个 `CardForm` 组件。

由于你将在不同的文件中实现逻辑代码，因此下面来看看所有需要完成的工作：

- 首先创建 `CardForm` 组件。
- 接着创建 `NewCard` 和 `EditCard` 组件。
- 接下来编辑 `App.js` 来设置新路由。
- 在 `KanbanBoardContainer` 类里面，你要创建方法来创建和编辑卡片。你将这些新方法作为 `props` 传递给 `NewCard` 和 `EditCard` 组件。

在你开始之前，确保你已经安装了 `React Router` 和 `History` 库：`npm install --save react-router@1.x.x history@1.x.x`。

### 1. CardForm 组件

如前所述，既然 `NewCard` 和 `EditCard` 组件的大部分 UI 都是相同的，所以你将创建 `CardForm` 组件，并在那两个组件中都使用它。`CardForm` 组件将包含一个表单(表单需要足够灵活，以能用于空白表单或者预填了现有卡片数据的表单)和一个覆盖效果。表单会以模式对话框的形式显示在看板上，覆盖效果用来将模式对话框后面的部分设置得更

暗一些。图 5-8 显示了想要的效果。

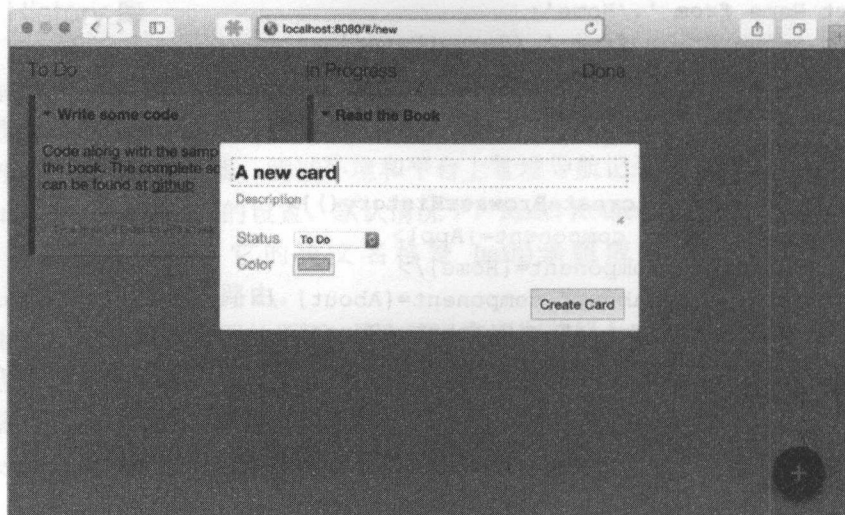


图 5-8 表单和样式

CardForm 组件将是一个单纯组件。它没有自己的 state。NewCard 或者 EditCard 组件需要向 CardForm 提供以下 props:

- 一个包含了要显示在表单上的数据的对象(下面把这个对象称为草稿卡片)。在 EditCard 的情况下, 这个对象将包含要被编辑的卡片的数据。在 NewCard 的情况下, 这个对象将包含一个新卡片所需的空白值或默认值。
- 提交按钮的标签。
- 处理表单域变更和表单提交的函数。
- 一个处理模式对话框关闭(当用户单击对话框之外的区域时就会关闭)的函数。

代码清单 5-23 显示了 CardForm 组件的源代码。

#### 代码清单 5-23: CardForm 组件的源代码

```
import React, {Component, PropTypes} from 'react';

class CardForm extends Component {
  handleChange(field, e){
    this.props.handleChange(field, e.target.value);
  }

  handleClose(e) {
    e.preventDefault();
    this.props.handleClose();
  }

  render() {
    return (
      <div>
```

```

<div className="card big">
  <form onSubmit={this.props.handleSubmit.bind(this)}>
    <input type='text'
      value={this.props.draftCard.title}
      onChange={this.handleChange.bind(this, 'title')}
      placeholder="Title"
      required={true}
      autoFocus={true} />
    <textarea value={this.props.draftCard.description}
      onChange={this.handleChange.bind(this, 'description')}
      placeholder="Description"
      required={true} />
    <label htmlFor="status">Status</label>
    <select id="status"
      value={this.props.draftCard.status}
      onChange={this.handleChange.bind(this, 'status')}>
      <option value="todo">To Do</option>
      <option value="in-progress">In Progress</option>
      <option value="done">Done</option>
    </select>
    <br />
    <label htmlFor="color">Color</label>
    <input id="color"
      value={this.props.draftCard.color}
      onChange={this.handleChange.bind(this, 'color')}
      type="color"
      defaultValue="#ff0000" />

    <div className='actions'>
      <button type="submit">{this.props.buttonLabel}</button>
    </div>
  </form>
</div>
<div className="overlay" onClick={this.handleClick.bind(this)}>
</div>
</div>
);
}
}

```

```

CardForm.propTypes = {
  buttonLabel: PropTypes.string.isRequired,
  draftCard: PropTypes.shape({
    title: PropTypes.string,
    description: PropTypes.string,
    status: PropTypes.string,
    color: PropTypes.string
  }).isRequired,
  handleChange: PropTypes.func.isRequired,

```

```

    handleSubmit: PropTypes.func.isRequired,
    handleClose: PropTypes.func.isRequired,
  }

```

```
export default CardForm;
```

接着，为创建图 5-8 所显示的结果，你需要为表单和覆盖效果添加一些额外样式。代码清单 5-24 显示了这些额外样式。

#### 代码清单 5-24：额外的 CSS 样式

```

.overlay {
  position: absolute;
  width: 100%;
  height: 100%;
  top: 0; left: 0; bottom: 0; right: 0;
  z-index: 2;
  background-color: rgba(0, 0, 0, 0.6);
}

```

```

.card.big {
  position: absolute;
  width: 450px;
  height: 200px;
  margin: auto;
  padding: 15px;
  top: 0; left: 0; bottom: 100px; right: 0;
  z-index: 3;
}

```

```

.card.big input[type=text], textarea {
  width : 100%;
  margin: 3px 0;
  font-size: 13px;
  border: none;
}

```

```

.card.big input[type=text] {
  font-size: 20px;
  font-weight: bold;
}

```

```

.card.big input[type=text]:focus,
.card.big textarea:focus {
  outline: dashed thin #999;
  outline-offset: 2px;
}

```

```

.card.big label {

```

```
margin: 3px 0 7px 3px;
color: #a7a7a7;
display: inline-block;
width: 60px;
}
```

```
.actions {
  margin-top: 10px;
  text-align: right;
}
```

```
.card.big button {
  font-size: 14px;
  padding: 8px;
}
```

## 2. NewCard 和 EditCard 组件

下面接着实现 NewCard 和 EditCard 组件。它们有许多共同点：两个组件都将包含卡片的草稿数据，都会渲染 CardForm，都将 state 以回调方式传递给 CardForm，并保存新增的或修改后的卡片。

首先创建 NewCard 组件，代码清单 5-25 显示了它的源代码。

代码清单 5-25: NewCard.js 组件最后的代码

```
import React, {Component, PropTypes} from 'react';
import CardForm from './CardForm'
```

```
class NewCard extends Component{
```

```
  componentWillMount(){
```

```
    this.setState({
      id: Date.now(),
      title: '',
      description: '',
      status: 'todo',
      color: '#c9c9c9',
      tasks: []
    });
```

```
  handleChange(field, value){
    this.setState({[field]: value});
  }
```

```
  handleSubmit(e){
    e.preventDefault();
    this.props.cardCallbacks.addCard(this.state);
    this.props.history.pushState(null, '/');
  }
```



```

    }

    handleClose(e) {
      this.props.history.pushState(null, '/');
    }

    render() {
      return (
        <CardForm draftCard={this.state}
          buttonLabel="Create Card"
          handleChange={this.handleChange.bind(this)}
          handleSubmit={this.handleSubmit.bind(this)}
          handleClose={this.handleClose.bind(this)} />
      );
    }
  }

  NewCard.propTypes = {
    cardCallbacks: PropTypes.object,
  };

export default NewCard;

```

上面的代码中有一些值得注意的事项:

- 当组件加载时, 它将组件的 `state` 设置为一个空的 `card` 草稿对象, 里面包含了一些默认值和一个临时 ID(临时 ID 基于当前的时间来创建)。这个 `card` 草稿对象的初始值, 和你对 `state` 进行的任何修改, 会显示在 `CardForm` 组件中。
- 当用户提交表单时, 你通过调用从 `KanbanBoardContainer` 作为 `props` 传递下来的 `cardCallbacks.addCard` 函数, 来保存新卡片的数据。

那就是 `NewCard` 组件的全部了。下面接着创建大同小异的 `EditCard` 组件, 这个组件和 `NewCard` 的区别是它会从路由接收一个 `card_id` 查询参数。使用这个卡片的 `id`, 你可以筛选卡片的信息, 并将用户想要编辑的卡片数据保存到它的 `state` 中, 如代码清单 5-26 所示。

#### 代码清单 5-26: `EditCard.js` 组件的完整代码

```

import React, {Component, PropTypes} from 'react';
import CardForm from './CardForm';

class EditCard extends Component {

  componentWillMount() {
    let card = this.props.cards.find((card) => card.id ==
      this.props.params.card_id);
    this.setState({...card});
  }
}

```

```

handleChange(field, value){
  this.setState({[field]: value});
}

handleSubmit(e){
  e.preventDefault();
  this.props.cardCallbacks.updateCard(this.state);
  this.props.history.pushState(null, '/');
}

handleClose(e){
  this.props.history.pushState(null, '/');
}

render(){
  return (
    <CardForm draftCard={this.state}
      buttonLabel="Edit Card"
      handleChange={this.handleChange.bind(this)}
      handleSubmit={this.handleSubmit.bind(this)}
      handleClose={this.handleClose.bind(this)} />
  )
}

EditCard.propTypes = {
  cardCallbacks: PropTypes.object,
}

export default EditCard;

```

### 3. 设置路由

下面先将注意力从 KanbanBoardContainer 移开一会儿,先在 App.js 文件中设置路由。这会帮助你更好地理解稍后将如何修改 KanbanBoardContainer。你将创建三个路由:一个指向 KanbanBoard 的 index 路由,一个和 NewCard 组件关联的 new 路由,以及一个和 EditCard 组件关联的 edit/:card\_id 路由。要指出的一点是, new 和 edit/:card\_id 路由会嵌套在 KanbanBoard 里面。这样你就可以查看 KanbanBoard 以及表单后面的卡片;你不希望表单成为一个完全孤立的区域。代码清单 5-27 显示了包含这些路由的 App.js 文件。

代码清单 5-27: App.js 里面定义的路由

```

import React from 'react';
import { render } from 'react-dom';
import { Router, Route } from 'react-router';
import createBrowserHistory from 'history/lib/createBrowserHistory';
import KanbanBoardContainer from './KanbanBoardContainer';
import KanbanBoard from './KanbanBoard';

```

```

import EditCard from './EditCard';
import NewCard from './NewCard';

render((
  <Router history={createBrowserHistory()}>
    <Route component={KanbanBoardContainer}>
      <Route path="/" component={KanbanBoard}>
        <Route path="new" component={NewCard} />
        <Route path="edit/:card_id" component={EditCard} />
      </Route>
    </Route>
  </Router>
), document.getElementById('root'));

```

#### 4. 在 KanbanBoardContainer 上创建回调函数并渲染子组件

在 KanbanBoardContainer 组件上要做的第一件事情，就是创建两个方法，从而创建和更新卡片。它们和处理任务数据的方法类似。代码清单 5-28 显示了 addCard 方法，代码清单 5-29 显示了 updateCard 方法。

##### 代码清单 5-28: KanbanBoardContainer 中的 addCard 方法

```

addCard(card) {
  // Keep a reference to the original state prior to the mutations
  // in case we need to revert the optimistic changes in the UI
  let prevState = this.state;

  // Add a temporary ID to the card
  if(card.id===null){
    let card = Object.assign({}, card, {id:Date.now()});
  }

  // Create a new object and push the new card to the array of cards
  let nextState = update(this.state.cards, { $push: [card] });

  // set the component state to the mutated object
  this.setState({cards:nextState});

  // Call the API to add the card on the server
  fetch(`${API_URL}/cards`, {
    method: 'post',
    headers: API_HEADERS,
    body: JSON.stringify(card)
  })
  .then((response) => {
    if(response.ok){
      return response.json()
    } else {
      // Throw an error if server response wasn't 'ok'

```

```

    // so we can revert back the optimistic changes
    // made to the UI.
    throw new Error("Server response wasn't OK")
  }
})
.then((responseData) => {
  // When the server returns the definitive ID
  // used for the new Card on the server, update it on React
  card.id=responseData.id
  this.setState({cards:nextState});
})
.catch((error) => {
  this.setState(prevState);
});
}

```

### 代码清单 5-29: KanbanBoardContainer 中的 updateCard 方法

```

updateCard(card) {
  // Keep a reference to the original state prior to the mutations
  // in case we need to revert the optimistic changes in the UI
  let prevState = this.state;

  // Find the index of the card
  let cardIndex = this.state.cards.findIndex((c)=>c.id == card.id);

  // Using the $set command, we will change the whole card
  let nextState = update(
    this.state.cards, {
      [cardIndex]: { $set: card }
    });
  // set the component state to the mutated object
  this.setState({cards:nextState});

  // Call the API to update the card on the server
  fetch(`${API_URL}/cards/${card.id}`, {
    method: 'put',
    headers: API_HEADERS,
    body: JSON.stringify(card)
  })
  .then((response) => {
    if(!response.ok) {
      // Throw an error if server response wasn't 'ok'
      // so we can revert back the optimistic changes
      // made to the UI.
      throw new Error("Server response wasn't OK")
    }
  })
  .catch((error) => {
    console.error("Fetch error:",error)
  })
}

```

```

    this.setState(prevState);
  });
}

```

最后，需要更新 `render` 方法。你并未手工渲染 `KanbanBoard`；它会被 `Router` 所注入。那样做的问题是，对于 `Router` 作为 `children` 来传递的组件，无法向其添加新的 `props`，但解决方案其实非常简单。如之前所见，你将克隆由 `Router` 所注入的 `props.children` 并添加新的 `props`，如代码清单 5-30 所示。

代码清单 5-30：克隆 `props.children` 并将卡片列表和回调函数作为 `props` 传入

```

render() {
  let kanbanBoard = this.props.children && React.cloneElement(
    this.props.children, {
      cards: this.state.cards,
      taskCallbacks: {
        toggle: this.toggleTask.bind(this),
        delete: this.deleteTask.bind(this),
        add: this.addTask.bind(this)
      },
      cardCallbacks: {
        addCard: this.addCard.bind(this),
        updateCard: this.updateCard.bind(this),
        updateStatus: this.updateCardStatus.bind(this),
        updatePosition: throttle(this.updateCardPosition.bind(this), 500),
        persistMove: this.persistCardMove.bind(this)
      }
    }
  );

  return kanbanBoard;
}

```

## 5. 在 `KanbanBoard` 组件中渲染卡片表单

我们已在 `App.js` 文件中配置了路由，将 `NewCard` 和 `EditCard` 组件设置成 `KanbanBoard` 的子组件。当用户访问路由 `/new` 或者 `/edit` 时，`router` 会在 `KanbanBoard` 中将相应组件作为 `props.children` 注入。你现在需要编辑 `KanbanBoard` 组件来克隆 `children`（以添加新 `props`，如卡片的列表和卡片的回调函数）并渲染它。这和你刚才在 `KanbanBoardContainer` 组件中做的事情大同小异。代码清单 5-31 显示了新的 `render` 函数。

代码清单 5-31：在 `KanbanBoard` 组件中渲染 `NewCard` 和 `EditCard`

```

render() {
  let cardModal=this.props.children && React.cloneElement(
    this.props.children, {
      cards: this.props.cards,
      cardCallbacks: this.props.cardCallbacks
    }
  );
}

```

```

return (
  <div className="app">
    <List ... />
    <List ... />
    <List ... />

    {cardModal}

  </div>
)
}

```

## 6. 最后一步：路由转换

你的应用程序的各个组件已经都连接起来，可正常运作了。如果你手工输入/new 路由或/edit/:card\_id(这里要使用一个正确的卡片 id)路由，一切将如期正常工作，但用户体验不是太好。因此让我们来处理路由的转换，先从/new 路由开始。

你将在 KanbanBoard 组件里面(使用 Link 组件)创建一个新链接，指向/new 路由。如代码清单 5-32 所示。

代码清单 5-32：在 KanbanBoard 组件上指向 new 路由的链接

```

import React, { Component, PropTypes } from 'react';
import { DragDropContext } from 'react-dnd';
import HTML5Backend from 'react-dnd-html5-backend';
import List from './List';
import { Link } from 'react-router';

class KanbanBoard extends Component {
  render() {
    let cardModal=this.props.children && React.cloneElement(
      this.props.children, {
        cards: this.props.cards,
        cardCallbacks: this.props.cardCallbacks
      });

    return (
      <div className="app">
        <Link to="/new" className="float-button">+</Link>

        <List ... />
        <List ... />
        <List ... />

        {cardModal}

      </div>
    )
  }
}

```



```
    }  
  }  
  KanbanBoard.propTypes = {...}  
  
  export default DragDropContext(HTML5Backend)(KanbanBoard);
```

你还将在链接上添加一些样式。样式是在屏幕的右下角的绝对定位位置，放置一个圆钮，如代码清单 5-33 和图 5-9 所示。

代码清单 5-33: 给按钮设置的 CSS 样式

```
.float-button {  
  position: absolute;  
  height: 56px;  
  width: 56px;  
  z-index: 2;  
  right: 20px;  
  bottom: 20px;  
  background: #D43A2F;  
  color: white;  
  border-radius: 100%;  
  font-size: 34px;  
  text-align: center;  
  text-decoration: none;  
  line-height: 50px;  
  box-shadow: 0 5px 10px rgba(0, 0, 0, 0.5);  
}
```

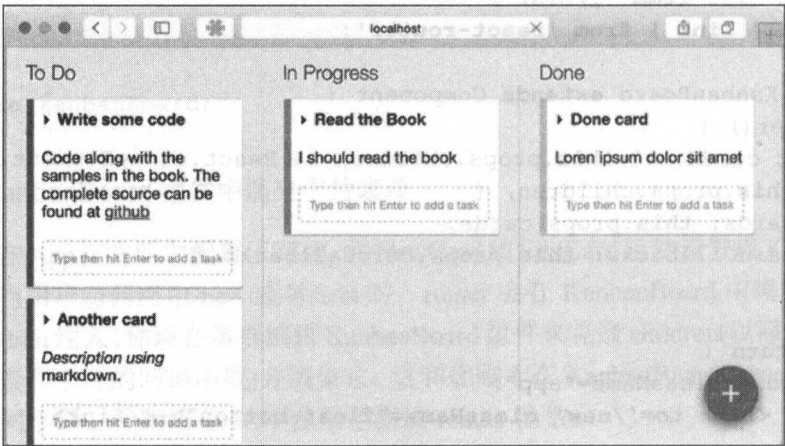


图 5-9 添加新卡片的按钮

最后，也为卡片添加一个 Link 组件。与新建按钮类似，也要给它应用一些样式，让它看起来和普通链接不一样。你将使用一些 CSS 技巧，来让链接只有在用户将鼠标放在卡片上时才显示出来。代码清单 5-34 显示了 Card 组件要使用的代码，代码清单 5-35 显示了为这个元素所添加的额外样式，图 5-10 显示了最终效果。

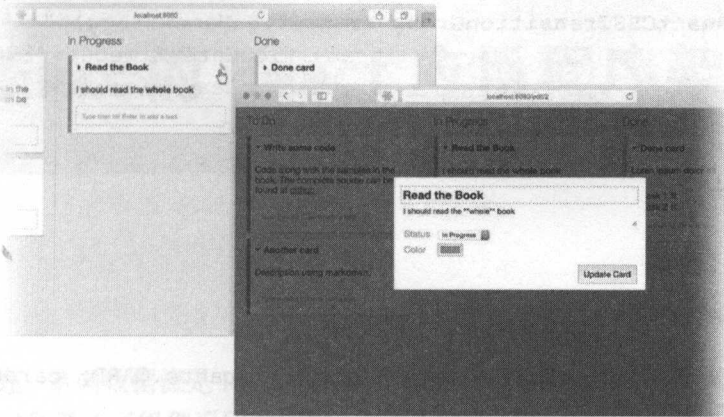


图 5-10 Card 组件上的 Edit 按钮

## 代码清单 5-34: Card 组件上的 Link 组件

```
import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from './constants';
import CheckList from './CheckList';
import {Link} from 'react-router';

let titlePropType = (props, propName, componentName) => {...};
const cardDragSpec = {...};
const cardDropSpec = {...};
let collectDrag = (connect, monitor) => {...};
let collectDrop = (connect, monitor) => {...};

class Card extends Component {
  constructor() {...}
  toggleDetails() {...}
  render() {
    const { connectDragSource, connectDropTarget } = this.props;

    let cardDetails;
    if (this.state.showDetails) {...};
    let sideColor = {...};

    return connectDropTarget(connectDragSource(
      <div className="card">
        <div style={sideColor}/>
        <div className="card_edit"><Link to=
          {'/edit/'+this.props.id}>&#9998;</Link></div>
        { /* &#9998; is the HTML entity for the utf-8 pencil character (✎) */ }
        <div className={...} onClick={...}>
          {this.props.title}
        </div>
      </div>
    ))
```

```

    <ReactCSSTransitionGroup transitionName="toggle"
                          transitionEnterTimeout={250}
                          transitionLeaveTimeout={250} >
      {cardDetails}
    </ ReactCSSTransitionGroup>
  </div>
  ));
}
}
Card.propTypes = {...};

const dragHighOrderCard = DragSource(constants.CARD, cardDragSpec,
  collectDrag)(Card);
const dragDropHighOrderCard = DropTarget(constants.CARD, cardDropSpec,
  collectDrop)(dragHighOrderCard);

export default dragDropHighOrderCard

```

代码清单 5-35：为 Card 组件上的编辑链接所准备的额外 CSS 样式

```

.card__edit{
  position: absolute;
  top:10px;
  right: 10px;
  opacity: 0;
  transition: opacity .25s ease-in;
}
.card:hover .card__edit{
  opacity: 1;
}
.card__edit a{
  text-decoration: none;
  color: #999;
  font-size: 17px;
}

```

## 5.3 本章小结

本章讨论了路由。首先，手工打造了一个基本的路由功能，并理解了由嵌套路由所引发的复杂性，接着学习了如何使用 React 社区中最常用的库之一的 React Router 库。你看到了如何设置嵌套路由和一个默认的首页路由，如何通过路由向一个组件传递参数，如何在一个组件中直接向它的子组件传递属性。之后还学习了如何使用 History 对象，通过代码进行路由转换。

你的看板应用程序相对于它的第一个版本，已经变得越来越大，功能也越来越丰富，同时也面临越来越多的问题。在第 6 章将要学习 Flux，这是一个协助 React 来更好地组织你的项目的应用程序架构。

# 结合 Flux 的 React 应用程序架构

如前所述，单向数据流是 React 的核心理念之一，即以 props 的形式从父组件流向子组件。当父组件需要子组件返回数据时，就可以像传递 props 一样传递一个回调函数。

这种单向数据流带来了清晰明了并且易于理解的代码。你可以从头到尾跟踪一个 React 应用程序，来观察更改后的代码是怎么样执行的。

即便这种架构模式具备许多优势，它还是面临一些挑战。React 应用程序常常会拥有许多层级，顶层组件扮演了容器的角色，而许多纯粹的组件则更像是界面树上的叶节点。state 位于这一层级的最高处，你创建的回调函数最终需要作为 props 向下传递，有时还会在重复并容易出错的任务中向下传递许多层级。

React Router 的共同作者和著名社区成员 Ryan Florence 这样形容这种将数据和回调函数以 props 形式向下传递许多层级的行为：钻取你的应用程序。如果你有许多嵌套组件，你就有大量的钻孔活儿要干，而且如果你想要重构(移动某些组件)，之前完成的钻取工作就必须完整地重做一遍。

让我来澄清一下：使用嵌套的 React 组件是构造 UI 的绝佳方式。它减少了复杂度、带来了关注点分离、并让代码更易于扩展和维护。由于 React 的构建理念是响应式渲染，每次当组件 state 或 props 发生变化时，React 都会更新 DOM(使用它的虚拟 DOM 实现来计算所需的最小变动量)。给你带来的则是非常简单的开发思维和卓越性能。

这里想要解决的是这样一个实际问题，当应用程序规模逐渐变大时，如何将数据(以及更重要的回调函数)带入到嵌套组件中，并能通过回调函数来操作这些组件的数据？这就是引入 Flux 的原因。

## 6.1 什么是 Flux

Flux 是 Facebook 创建的一份针对构建 Web 应用程序的架构指引。虽然它既不是 React 的一部分，也并不是专门为 React 所创建的，但它和 React 却十分般配。Flux 的要点是允许应用程序中存在单向数据流。它由三个基本部分组成：Action、Store 和 Dispatcher。让我们来看一看这三个部分。

### 6.1.1 Store

如前所述，你正在尝试解决的要点之一是如何将数据带入到应用程序的每个组件。



我们理想的世界如图 6-1 所示。数据完全和组件分离，但你想要在数据发生变化时通知组件，这样它就可以重新渲染了。

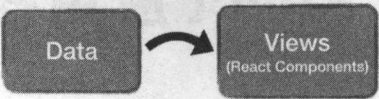


图 6-1 理想的世界

这就是Store要做的事情。Store存放着所有应用程序状态(包括数据甚至UI状态)，并会在状态发生变化时分发事件。View(React组件)订阅了包含它们所需的数据的Store，当数据发生变化时，它就会重新渲染自身，如图 6-2 所示。

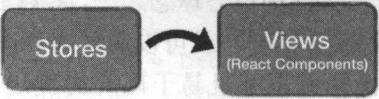


图 6-2 View 重新渲染自身

Store 的一个重要特征是它们近似于黑盒；它们公开了公共 getter 来访问数据，但谁也没有办法插入、更新或修改它们的数据，View 和 Flux 的其他部分都没有这个能力。只有 Store 自身可以操作其数据。

如果你了解 MVC 范式，会发现 Store 和 Model 惊人地相似，但它们之间的主要区别还是 Store 只有 getter，没有人可以设置 Store 中的值。

但是，如果应用程序的其他部分都没有办法修改 Store 中的数据，系统该通过什么来通知 Store 更新其数据呢？答案就是 Action。

6.1.2 Action

Action 可以不严谨地定义为“应用程序中发生的事情”。Action 几乎可以由应用程序的任何部分创建，通常会由用户交互(比如单击一个按钮、留下一段评论、请求搜索结果等)发起，但也可以当作 AJAX 请求、计时器和 Web Socket 事件等的结果。

每个Action都包含一个type(它的唯一名称)和一个可选的payload。当被分发后，Action 就会到达Store，这就是Store得知它应当更新其数据的方法。如图 6-3 所示。

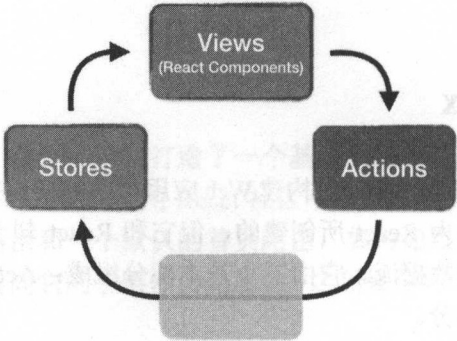


图 6-3 Store 更新其数据

实际上，这几乎就是 Flux 的全部了：React 组件创建一个 Action(假设每当用户在一

个文本域中输入一个名称后); Action 到达 Store; 对这一特定 Action 有兴趣的 Store 更新其数据并分发更改的事件; 最后, View 响应了 Store 的事件, 使用最新的数据重新进行渲染。但这幅图还缺一块: Dispatcher。

6.1.3 Dispatcher

Dispatcher 负责协调 Action 传递到 Store 的过程, 并确保以正确顺序执行 Store 的 Action 处理程序。如图 6-4 所示。

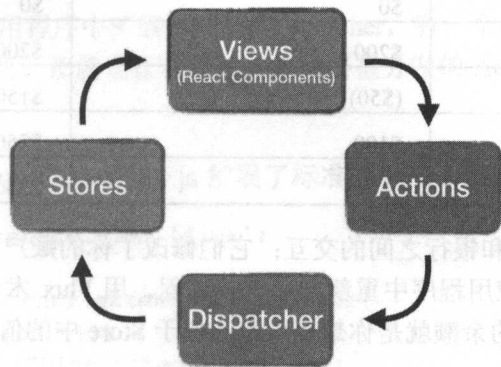


图 6-4 Dispatcher 工作流

尽管 Dispatcher 是 Flux 架构的基本组成之一, 但你并不需要过多考虑它。你所需要做的仅是创建一个实例, 然后使用它; 其余的就交由 Dispatcher 的实现去处理。

6.2 假想的简单 Flux 应用程序

在复杂应用程序中使用 Flux 时, 它有助于保持代码更易于理解、维护和成长。它确实降低了复杂度, 由此产生的结果是, 在许多情形中还减少了项目中的代码行数(尽管代码行数并不是恰当的代码复杂度的度量标准)。不过在第一个示例项目中并不会出现这种情况, 因为使用 Flux 实际上增加了构建它所需的代码数量。这是因为第一个示例的目的是帮助你理解 Flux 应用程序中的所有元素。对新手而言, Flux 可能有些复杂, 所以你会从一个非常基本的项目开始, 这个项目几乎没有 UI(我的意思是没有特别复杂的 UI), 它旨在直观且合规地使用 Flux+React 应用程序中的所有元素。在下一节中, 你将会接触到一个完整的现实世界中的示例。

银行账户应用程序

Jeremy Morrell 在他的演讲《那些忘记过去的人注定要调试它》(<https://speakerdeck.com/jmorrell/jsconf-uy-flux-those-who-forget-the-past-dot-dot-dot-1>)中第一次使用银行账户来形容 Flux Action 和 Store, 本书在使用时得到了他的允许。

银行账户由两个部分组成: 交易和余额。每一次交易时都会更新余额, 如表 6-1 和表 6-2 所示。



表 6-1 第一次交易初始化了余额

交易	金额	余额
创建账户	\$0	\$0

表 6-2 每一次交易都会更新余额

交易	金额	余额
创建账户	\$0	\$0
存款	\$200	\$200
取款	(\$50)	\$150
存款	\$100	\$250
		\$250

这些交易代表了你和银行之间的交互；它们修改了你的账户状态。

你将在一个 Flux 应用程序中重新创建这一过程。用 Flux 术语讲，表左边的交易就是你的 Action，表右边的余额就是你想要跟踪的位于 Store 中的值。你的示例应用程序结构将包含：

- 一个 constants.js 文件(因为所有 Action 都应当具备一个应用程序级别的唯一标识名，你会将这些名称存储为常量)。
- 标准的 AppDispatcher.js。
- BankActions.js，它包含了三个 Action 创建器：CreateAccount、depositIntoAccount 和 WithdrawFromAccount。我们把这些方法称为“Action 创建器”是因为 Action 其实就是对象。由于没想到更好的名字，这些创建并分发了 Action 的方法就被称为 Action 创建器。
- BankBalanceStore.js 将跟踪用户的余额。
- 最后，app.js 文件包含了一个 UI 组件，你会在本项目中用到它。

首先创建一个新项目，然后使用 npm(npm install --save flux)来安装 Flux 库。接下来会逐一介绍项目中的每个文件。

应用程序常量

让我们从定义常量文件开始着手。你需要三个常量，用于在整个应用中唯一地标识你的 Action：创建账户、存入账户和从账户提取。

代码清单 6-1 展示了相关代码。

代码清单 6-1: constants.js 文件

```
export default{
  CREATED_ACCOUNT:'created account',
  WITHDREW_FROM_ACCOUNT:'withdrew from account',
  DEPOSITED_INTO_ACCOUNT:'deposited into account'
};
```

## Dispatcher

接下来, 让我们定义你的应用程序 Dispatcher。如之前所述, 对此你并不需要考虑过多。就这一点来说, 你的 AppDispatcher 文件可以简单到只是实例化了一个 Flux Dispatcher 而已。

```
import {Dispatcher} from 'flux';
export default new Dispatcher();
```

但你确实能在应用程序中扩展标准的 Dispatcher, 有一个示例可以帮助你更好地理解 Dispatcher 这一角色, 那就是让它将每一个将要被分发的 Action 记录下来, 如代码清单 6-2 所示。

代码清单 6-2: AppDispatcher.js 扩展了标准的 Flux Dispatcher 来记录每一次分发

```
import {Dispatcher} from 'flux';

class AppDispatcher extends Dispatcher{
  dispatch(action = {}) {
    console.log("Dispatched", action);
    super.dispatch(action);
  }
}

export default new AppDispatcher();
```

## Action 创建器

让我们接着完成这个假想的银行应用程序, 接下来定义一些为应用程序生成 Action 的函数。请记住, 在 Flux 应用程序中, Action 仅是一个包含了 type 和可选数据载荷的对象而已。由于缺少更好的术语, 我们把这些定义并分发了 Action 的函数称为 Action 创建器, 你可创建一个包含三个 Action 创建器(创建账户、存款和取款)的 JavaScript 文件, 如代码清单 6-3 所示。

代码清单 6-3: BankActions.js 文件

```
import AppDispatcher from './AppDispatcher';
import bankConstants from './constants';

let BankActions = {
  /**
   * Create an account with an empty value
   */
  createAccount() {
    AppDispatcher.dispatch({
      type: bankConstants.CREATED_ACCOUNT,
      ammount: 0
    });
  }
};
```

```

    },

    /**
     * @param{number} ammount to whithdraw
     */
    depositIntoAccount(ammount) {
      AppDispatcher.dispatch({
        type: bankConstants.DEPOSITED_INTO_ACCOUNT,
        ammount: ammount
      });
    },

    /**
     * @param{number} ammount to whithdraw
     */
    withdrawFromAccount(ammount) {
      AppDispatcher.dispatch({
        type: bankConstants.WITHDREW_FROM_ACCOUNT,
        ammount: ammount
      });
    }
  };

  export default BankActions;

```

## Store

按照顺序, 我们该定义 `BankBalanceStore` 文件了。在一个 Flux 应用程序中, Store 拥有 state, 并通过 Dispatcher 注册了自己。每次分发 Action 时, 所有 Store 都会被调用, 并且可以决定它们是否关心该 Action。如果有一个 Store 器表示关心, 它就会更改它的内部 state, 并发送一个事件, 这样 View 就能得到 Store 发生变化的通知了。

要发送事件, 你需要一个来自 npm 的事件发射器包。Node.js 拥有一个默认的事件发射器, 但浏览器并不支持它。npm 中有各种各样的包在浏览器中重新实现了 node 的事件系统; 甚至 Facebook 也有一个开源实现, 它是一个简单的实现, 优先考虑了速度和简单性。让我们来使用它: `npm install --save fbemitter`。

现在开始为 `BankAccountStore` 编写基本结构, 创建一个事件发射器实例, 并提供一个 `addListener` 方法来订阅 Store 的变化事件。你还要导入应用程序 Dispatcher 然后注册 Store, 向其提供一个回调函数, 每次分发 Action 时都会触发该回调函数。代码清单 6-4 展示了这部分代码。

### 代码清单 6-4: 使用普通 JavaScript 对象的基础 Store

```

import {EventEmitter} from 'fbemitter';
import AppDispatcher from './AppDispatcher';
import bankConstants from './constants';

```

```

const CHANGE_EVENT = 'change';
let __emitter = new EventEmitter();

let BankBalanceStore = {
  addListener: (callback) => {
    return __emitter.addListener(CHANGE_EVENT, callback);
  },
};

BankBalanceStore.dispatchToken = AppDispatcher.register((action) => {
  switch (action.type) {
    // ...
  }
});

export default BankBalanceStore;

```

注意，在代码里，你调用了 `Dispatcher` 的注册方法，并传入一个回调函数。每次分发时，都会调用这个函数，所以你有机会决定在特定 `Action type` 被分发时，`Store` 是否要做某些事情。

此外，`Dispatcher` 的 `register` 方法返回了一个分发令牌：这是一个可用来协调 `Store` 更新顺序的标识符，你将在本章后面看到这一用途。

按照顺序，还需要做两件事情：创建一个变量来存储账户余额(和一个访问该值的 `getter` 方法)，以及用来响应 `Action CREATE_ACCOUNT`、`DEPOSIT_INTO_ACCOUNT` 和 `WITHDRAW_FROM_ACCOUNT` 的 `switch` 语句。注意，在更改了账户余额的内部值后，需要手工发送一个变化事件。完整的代码如代码清单 6-5 所示。

代码清单 6-5: `BankBalanceStore.js` 的完整代码

```

import {EventEmitter} from 'fbemitter';
import AppDispatcher from './AppDispatcher';
import bankConstants from './constants';

const CHANGE_EVENT = 'change';
let __emitter = new EventEmitter();
let balance = 0;

let BankBalanceStore = {
  getState() {
    return balance;
  },
  addListener(callback) {
    return __emitter.addListener(CHANGE_EVENT, callback);
  }
};

```

```

    });

    BankBalanceStore.dispatchToken=AppDispatcher.register((action)=>{
      switch (action.type) {
        case bankConstants.CREATED_ACCOUNT:
          balance =0;
          __emitter.emit(CHANGE_EVENT);
          break;

        case bankConstants.DEPOSITED_INTO_ACCOUNT:
          balance = balance +action.ammount;
          __emitter.emit(CHANGE_EVENT);
          break;

        case bankConstants.WITHDREW_FROM_ACCOUNT:
          balance = balance -action.ammount;
          __emitter.emit(CHANGE_EVENT);
          break;
      }
    });

    export default BankBalanceStore;
  }

```

### UI 组件

最后，需要一些UI。你的App.js文件将导入Store和Action。它将显示由Store控制的余额，并在用户单击了取款或存款按钮后调用Action创建器。

让我们实现这一部分，先从Store开始。如代码清单6-6所示，在类构造函数中，定义了包含balance键的私有state。该键所对应的值来自BankBalanceStore(BankBalanceStore.getState())。按照顺序，你使用生命周期函数componentDidMount和componentWillUnmount来管理对BankBalanceStore中的变化的侦听。只要Store发生变化，handleStoreChange方法就会被调用，组件的state也会被更新(并且，正如你所知，state发生变化时，组件会重新渲染自身)。

代码清单 6-6：应用程序组件从 BankBalanceStore 获取其 state 的部分代码

```

import React,{ Component } from 'react';
import { render } from 'react-dom';
import BankBalanceStore from './BankBalanceStore';
import BankActions from './BankActions';

class App extends Component {
  constructor() {
    super(...arguments);
    BankActions.createAccount();
    this.state={

```

```

    balance: BankBalanceStore.getState()
  }
}

componentDidMount() {
  this.storeSubscription = BankBalanceStore.addListener(
    data => this.handleStoreChange (data));
}

componentWillUnmount() {
  this.storeSubscription.remove();
}

handleStoreChange() {
  this.setState({balance: BankBalanceStore.getState()});
}
}

```

按照顺序，我们来实现 `render` 函数。它包含一个文本域和两个按钮(取款和存款)。你还需要两个私有方法来处理这两个按钮的单击事件。这些方法只是简单地调用了 `Action` 创建器并清空了文本域而已。代码清单 6-7 展示了 `App.js` 的完整代码，作为补充，代码清单 6-8 展示了包含基本样式的 `CSS` 文件。

#### 代码清单 6-7：完整的应用程序组件

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import BankBalanceStore from './BankBalanceStore';
import BankActions from './BankActions';

class App extends Component {
  constructor() {
    super(...arguments);
    BankActions.createAccount();
    this.state = {
      balance: BankBalanceStore.getState()
    }
  }

  componentDidMount() {
    this.storeSubscription = BankBalanceStore.addListener(
      data => this.handleStoreChange(data));
  }

  componentWillUnmount() {
    this.storeSubscription.remove();
  }

  handleStoreChange() {

```



```

    this.setState({balance: BankBalanceStore.getState()});
  }

  deposit() {
    BankActions.depositIntoAccount(Number(this.refs.ammount.value));
    this.refs.ammount.value = '';
  }

  withdraw() {
    BankActions.withdrawFromAccount(Number(this.refs.ammount.value));
    this.refs.ammount.value = '';
  }

  render() {
    return (
      <div>
        <header>FluxTrust Bank</header>
        <h1>Your balance is ${((this.state.balance).toFixed(2))}</h1>
        <div className="atm">
          <input type="text" placeholder="Enter Ammount" ref="ammount" />
          <br />
          <button onClick={this.withdraw.bind(this)}>Withdraw</button>
          <button onClick={this.deposit.bind(this)}>Deposit</button>
        </div>
      </div>
    );
  }
}

render(<App />, document.getElementById('root'));

```

### 代码清单 6-8：假想银行应用程序的基础样式

```

body {
  margin:0;
  font:16px/1sans-serif;
  background-color:#318435;
  color:#fff;
  text-align:center;
}

header{
  width:100%;
  padding:15px;
  text-align:center;
  background-color:#000;
}

h1{
  font-size: 18px;
}

h2{

```

```

font-size: 16px;
}
.atm{
width: 200px;
height: 100px;
border-radius: 10px;
background-color: #000;
text-align: center;
margin: 10px auto 0 auto;
padding: 20px;
}
.atm input{
font-size: 25px;
width: 180px
}
.atm button{
margin: 5px;
padding: 20px;
}

```

如果完成了以上所有步骤，现在是时候尝试一下取款和存款操作了。请确保已经打开了浏览器控制台，这样就可以看到被 Dispatcher 记录下来的所有 Action 了，如图 6-5 所示。

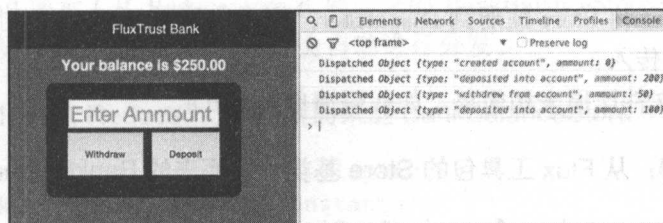


图 6-5 假想银行应用程序及其记录下来的 Action

## 6.3 Flux 工具包

从 2.1 版开始，Flux 库就包含了用来定义 Store 的基类，还有一个搭配容器组件一起使用的高阶函数——允许组件在相关的 Store 发生变化后自动更新其 state。这些实用程序的价值在于它们有助于减少应用程序中乏味的样板代码。

### 6.3.1 Flux Store 工具

Flux 工具包提供了三个用来实现 Store 的基类：Store、ReduceStore 和 MapStore。

- Store 是最简单的一个。它只是基础 Store 的一个轻量级包装。它有助于应付样板代码，但没有带来任何新的概念或功能。

- **ReduceStore** 是一种非常特殊的 **Store**。它的名字源于它使用化简函数来修改内部 **state** 这一事实。**Reducer** 是一个用来根据之前给定的 **state** 和 **Action** 来计算出新 **state** 的函数，与 **Array.prototype.reduce** 的功能类似。**ReduceStore** 中的 **state** 会被当作不可变量来处理，所以要注意只存储不可变的结构，或者以下范围内的值：
  - 单一原始值(字符串、布尔值或者数字)。
  - 一个原始值组成的数组，如 `[1,2,3,4]`。
  - 一个原始值组成的对象，如 `{name:'cassio', age:35}`。
  - 一个包含内嵌对象的对象，并将使用 **React** 不变性助手来操作这些对象。
- **MapStore** 是 **ReduceStore** 的一个变种，它包含一个额外的辅助函数来存储键值对(而非单一值)。

**ReduceStore**(以及 **MapStore**)的另一个亮点是你无须手动触发变化事件：每次分发前后都会自动比较 **state** 并发送变化事件。

让我们用上一个示例中的 **BankBalanceStore** 为例来演示 **Flux** 工具包中的 **Store** 基类。出于比较的目的，首先来看一看如何使用 **Flux** 工具包的 **Store** 来达到同样的效果。接下来，你还会使用 **ReduceStore** 来创建一个更加简洁的版本。

实际上使用 **Store** 基类的实现和你当前的 **BankBalanceStore** 几乎完全一样，只有两点主要区别：

- 不需要创建事件发射器的实例。
- 不需要手工将 **Store** 注册到 **Dispatcher**；而是在创建 **Store** 实例时，将 **Dispatcher** 作为参数传入。

最终结果是文件比原来稍简单，如代码清单 6-9 所示。

代码清单 6-9：从 **Flux** 工具包的 **Store** 基类扩展而来的 **BankBalanceStore** 版本

```
import AppDispatcher from './AppDispatcher';
import {Store} from 'flux/utils';
import bankConstants from './constants';

let balance = 0;

class BankBalanceStore extends Store {
  getState() {
    return __balance;
  }
  __onDispatch(action) {
    switch (action.type) {
      case bankConstants.CREATED_ACCOUNT:
        balance = 0;
        this.__emitChange();
        break;

      case bankConstants.DEPOSITED_INTO_ACCOUNT:
        balance = balance + action.ammount;
    }
  }
}
```

```

    this.__emitChange();
    break;

    case bankConstants.WITHDREW_FROM_ACCOUNT:
        balance = balance - action.ammount;
        this.__emitChange();
        break;
    }
}

export default new BankBalanceStore(AppDispatcher);

```

与使用 Store 相比, 使用 ReduceStore 的实现才真正精彩。除了更简洁外, 它对不可变数据结构的使用能让编程变得更具声明性(就像 React), 并且对许多其他领域(例如, 测试)都产生了积极影响。

让我们再实现一个新的 BankBalanceStore, 这一次使用 ReduceStore。要扩展 ReduceStore, 你的类需要实现两个方法: `getInitialState` 和 `reduce`。在 `getInitialState` 中, 你定义 Store 的初始 state; 而在 `reduce` 中, 你根据 Action 来修改 state。默认的 `getState` 方法已经定义好了, 不需要改写它(除非你不想以不可变量的形式处理 ReduceStore state, 这有悖于使用 ReduceStore 的初衷, 所以在实际中, 总是要将其作为不可变量来处理)。

代码清单 6-10 展示了从 ReduceStore 扩展而来的 BankBalanceStore 的完整代码。注意这里并不需要发送变化事件, 它们总是会进行自动分发。

#### 代码清单 6-10: 从 ReduceStore 扩展而来的 BankBalanceStore

```

import AppDispatcher from './AppDispatcher';
import bankConstants from './constants';
import {ReduceStore} from 'flux/utils';

class BankBalanceStore extends ReduceStore {
    getInitialState() {
        return 0;
    }

    reduce(state, action) {
        switch (action.type) {

            case bankConstants.CREATED_ACCOUNT:
                return 0;

            case bankConstants.DEPOSITED_INTO_ACCOUNT:
                return state + action.ammount;

            case bankConstants.WITHDREW_FROM_ACCOUNT:
                return state - action.ammount;

```

```

    default:
      return state;
  }
}

export default new BankBalanceStore(AppDispatcher);

```

### 6.3.2 容器组件高阶函数

你在第 3 章中学习了容器组件。它们用于分离和 UI 渲染无关的业务逻辑(比如数据获取)和与其对应的子组件。默认情况下,容器是纯粹的,这意味着当它们的 state 没有改变时,它们不会重新进行渲染。

提示:

注意,要使用 Flux 工具包中的高阶函数,容器组件就不能访问任何 props。这既是出于性能原因,也是为确保容器可被重用,并且无须在整个组件树中处理 props。

下面来看一看这个示例。你将修改应用程序组件,使其能够自动订阅 BankAccountStore 并在其发生变化时自动更新自己的 state。你会从删除不再需要的代码开始着手:你不必在构造函数中声明组件的初始 state。因为高阶组件会为你解决订阅和取消订阅 Store 的事宜,所以 componentDidMount 和 componentWillUnmount 生命周期方法也可以删掉了。出于同样的原因,你还能摆脱 handleStoreChange 方法。要使用高阶函数,你的容器组件必须实现两个类方法:calculateState(用来将 Store state 映射到组件私有 state)和 getStores(返回所有组件所侦听的 Store 数组)。注意容器高阶函数只适用于从 Flux 工具包的 Store 扩展而来的 Store。代码清单 6-11 展示了更新后的 App.js 组件,现在它比之前的版本小了 15%。

代码清单 6-11: 使用了 Flux 工具包中的容器高阶函数的应用程序组件

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import { Container } from 'flux/utils';
import BankBalanceStore from './BankBalanceStore';
import BankActions from './BankActions';

class App extends Component {
  constructor() {
    super(...arguments);
    BankActions.createAccount();
  }

  deposit() {
    BankActions.depositIntoAccount(Number(this.refs.ammount.value));
    this.refs.ammount.value = '';
  }
}

```

```

    }
  }

  withdraw() {
    BankActions.withdrawFromAccount(Number(this.refs.ammount.value));
    this.refs.ammount.value = '';
  }

  render() {
    return (
      <div>
        <header>FluxTrust Bank</header>
        <h1>Your balance is ${this.state.balance}.toFixed(2)}</h1>
        <div className="atm">
          <input type="text" placeholder="Enter Ammount" ref="ammount" />
          <br />
          <button onClick={this.withdraw.bind(this)}>Withdraw</button>
          <button onClick={this.deposit.bind(this)}>Deposit</button>
        </div>
      </div>
    );
  }
}

App.getStores = () => ([BankBalanceStore]);
App.calculateState = (prevState) => ({
  balance: BankBalanceStore.getState()});

const AppContainer = Container.create(App);

render(<AppContainer />, document.getElementById('root'));
```

## 6.4 异步 Flux

在任何较为复杂 JavaScript Web 应用程序中，你大概都需要和异步打交道。主要通过两种形式：在 Store 之间协调更新顺序以及异步获取数据。

### 6.4.1 waitFor：协调 Store 的更新顺序

在大型 Flux 项目中处理多个 Store 时，你可能会发现某个 Store 依赖于来自另一个 Store 的数据。Flux Dispatcher 提供了一个 `waitFor()` 方法来管理这种依赖关系；它能让 Store 在继续执行之前，先等待来自指定 Store 的回调被调用。

下面为假想银行应用程序增加一个奖励程序，该程序依赖的是用户的当前余额。你可以创建一个新的 `BankRewardsStore` 来处理当前用户在程序中的等级，由于程序单独依赖该用户的余额，所以 `BankRewardsStore` 的所有操作都必须先等待 `BankBalanceStore` 完成更新，然后再据此来更新自身。代码清单 6-12 展示了最终完成的 `BankRewardsStore`。



## 代码清单 6-12: BankRewards Store

```

import AppDispatcher from './AppDispatcher';
import BankBalanceStore from './BankBalanceStore';
import bankConstants from './constants';
import {ReduceStore} from 'flux/utils';

export default new BankRewardsStore(AppDispatcher);

class BankRewardsStore extends ReduceStore {
  getInitialState() {
    return 'Basic';
  }
  reduce(state, action){
    this.getDispatcher().waitFor([
      BankBalanceStore.getDispatchToken()
    ]);
    if (action.type===bankConstants.DEPOSITED_INTO_ACCOUNT||
      action.type===bankConstants.WITHDREW_FROM_ACCOUNT ) {
      let balance =BankBalanceStore.getState();
      if (balance < 5000)
        return 'Basic';
      else if (balance < 10000)
        return 'Silver';
      else if (balance < 50000)
        return 'Gold';
      else
        return 'Platinum';
    }
    return state;
  }
}

```

BankRewardsStore响应DEPOSIT\_INTO\_ACCOUNT和WITHDRAW\_FROM\_ACCOUNT这两种Action type的方法相同：获取当前余额，然后简单地根据余额来指派一个级别(余额少于5000美元的情况为Basic级别；余额在5000美元和10 000美元之间为Silver级别；余额超过50 000美元为Platinum级别)。

现在可以在你的主要组件中订阅这个 Store 并展示用户在奖励程序中的级别了。代码清单 6-13 展示了更新后的 App.js，图 6-6 则展示了更新后的应用程序。

## 代码清单 6-13: 更新后的 App.js 订阅了 BankRewardsStore

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import {Container} from 'flux/utils';
import BankBalanceStore from './BankBalanceStore';
import BankRewardsStore from './BankRewardsStore';
import BankActions from './BankActions';

class App extends Component {

```

```

constructor(){
  super(...arguments);
  BankActions.createAccount();
}

deposit() {
  BankActions.depositIntoAccount(Number(this.refs.ammount.value));
  this.refs.ammount.value = '';
}

withdraw() {
  BankActions.withdrawFromAccount(Number(this.refs.ammount.value));
  this.refs.ammount.value = '';
}

render() {
  return (
    <div>
      <header>FluxTrust Bank</header>
      <h1>Your balance is ${this.state.balance}.toFixed(2)}</h1>
      <h2>Your Points Rewards Tier is {this.state.rewardsTier}</h2>
      <div className="atm">
        <input type="text" placeholder="Enter Ammount" ref="ammount" />
        <br />
        <button onClick={this.withdraw.bind(this)}>Withdraw</button>
        <button onClick={this.deposit.bind(this)}>Deposit</button>
      </div>
    </div>
  );
}
}

App.getStores = () => ([BankBalanceStore, BankRewardsStore]);
App.calculateState = (prevState) => ({
  balance: BankBalanceStore.getState(),
  rewardsTier: BankRewardsStore.getState()
});

const AppContainer = Container.create(App);

render(<AppContainer />, document.getElementById('root'));

```

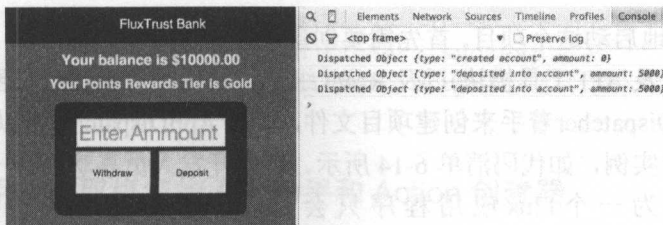


图 6-6 更新后的假想账号以及假想奖励程序

## 6.4.2 异步数据获取

到目前为止,你所看到的 Flux 常规用法都非常简单,然而也有一件并不直观的事情:处理异步请求的位置。你应该在哪里获取数据?如何让响应进入 Flux 数据流?

这个库没有硬性规定发起获取操作的特定位置,但来自社区的最佳实践推荐将所有请求和 API 调用封装到一个独立模块中(比如 APIutils.js 这样的文件)。API 工具可以从任意位置调用,但它们会总是发起异步请求,然后通过 Action 创建器来分发 Action(所以任何 Store 都可以选择是否进行响应)。

记住,在调用一个异步 API 时,会出现两个关键时刻:开始调用的时刻以及接收到回复(或者超时)的时刻。因此,API 工具模块应该总是分发至少三种不同类型的 Action:一个告知 Store 请求已经开始的 Action,一个告知 Store 请求成功完成的 Action,以及一个告知 Store 请求失败的 Action。

把和 API 的交互封装到一个单独的模块里并且分发不同的 Action 会带来许多好处,因为它将系统其余部分隔绝到异步执行之外。当 Action 被分发后,对于 Store 和组件而言,后续代码是以同步方式执行的,理解和推断起来也就更加轻松。

下面创建一个新的应用程序——机票网站——来例证这种方法。

## 6.5 AirCheap 应用程序

这个应用程序会在加载时立即获取机场代码清单,当用户填写了出发机场和到达机场后,应用程序会和 API 进行通信来获取机票价格。图 6-7 展示了可运作的应用程序。

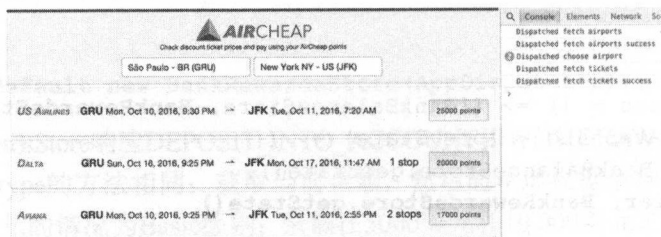


图 6-7 AirCheap 机票应用程序

### 6.5.1 搭建:项目组织和基本文件

为更有条理地启动这个项目,首先需要创建一些文件夹来存放 Flux 相关文件(Action 创建器、Store 以及 API 工具模块)以及 React 组件。初始的项目结构如图 6-8 所示。

下面从 AppDispatcher 着手来创建项目文件,记住,AppDispatcher 可以简单到只是 Flux Dispatcher 的一个实例,如代码清单 6-14 所示。有些开发人员喜欢创建一个 dispatchers 文件夹,但是因为一个 Flux 应用程序只会包含一个 Dispatcher,所以我们会把 AppDispatcher.js 文件保存到顶级 app 文件夹下。

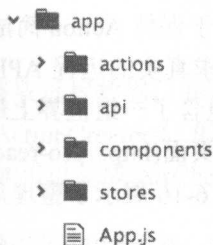


图 6-8 AirCheap 项目的应用程序文件夹结构

#### 代码清单 6-14: AppDispatcher.js 最简形式

```
import {Dispatcher} from 'flux';
export default new Dispatcher();
```

当然，你可以扩展 Dispatcher 的功能。比如你想要记录 Dispatcher 所分发的所有 Action，就可以覆盖 dispatch 方法，就像你在假想的银行账户应用程序中做的那样。

接下来创建 constants.js 文件，它只包含一些常量，我们会根据需要来扩充它们，但在这个示例中，你预先已经知道了将要用到的所有常量了：

- **FETCH\_AIRPORTS** 是一个 Action 名称，你会在应用程序启动时分发此 Action 来获取所有机场信息。由于这是一个异步操作，你还需要创建 **FETCH\_AIRPORTS\_SUCCESS** 和 **FETCH\_AIRPORTS\_ERROR** 常量来表示操作的成功或者错误状态。
- **CHOOSE\_AIRPORT** 是一个同步 Action 名称，该 Action 表示用户选择了一个机场(无论是出发地还是目的地)。
- **FETCH\_TICKETS** 是一个 Action 名称，你会在出发地和目的地都被选中后分发此 Action。这是一个异步数据获取操作，所以你还需要常量来表示获取操作的成功和错误状态：**FETCH\_TICKETS\_SUCCESS** 和 **FETCH\_TICKETS\_ERROR**。

代码清单 6-15 展示了最终的 constants.js 文件。

#### 代码清单 6-15: constants.js 文件

```
export default{
  FETCH_AIRPORTS:'fetch airports',
  FETCH_AIRPORTS_SUCCESS:'fetch airports success',
  FETCH_AIRPORTS_ERROR:'fetch airports error',
  CHOOSE_AIRPORT:'choose airport',
  FETCH_TICKETS:'fetch tickets',
  FETCH_TICKETS_SUCCESS:'fetch tickets success',
  FETCH_TICKETS_ERROR:'fetch tickets error'
};
```

### 6.5.2 创建用于获取机场的 API 助手和 Action 创建器

下面创建一个 API 助手来辅助获取机场和机票。正如之前所讨论的那样，创建一个

独立的助手模块来和 API 交互有助于保持 Action 简洁。由于这只是一个示例应用程序，为简单起见，API 助手方法不会请求真实的远程 API，而是会从你的 public 文件夹中加载一个静态 json 文件，这个文件包含了一组世界上最大的机场。你可以从 Apress 网站 ([www.apress.com](http://www.apress.com)) 或本书的 GitHub 页面 (<http://pro-react.github.io>) 下载 airports.json 文件和本项目的其他公共资源。代码清单 6-16 展示了整理后的 airports.json 文件。

代码清单 6-16: 整理后的 public/airports.json 文件

```
[
  {"code": "ATL", "city": "Atlanta GA", "country": "US"},
  {"code": "LHR", "city": "London", "country": "GB"},
  {"code": "JFK", "city": "New York NY", "country": "US"},
  {"code": "ORD", "city": "Chicago IL", "country": "US"},
  {"code": "HND", "city": "Tokyo", "country": "JP"},
  {"code": "LAX", "city": "Los Angeles CA", "country": "US"},
  {"code": "CDG", "city": "Paris", "country": "FR"},
  {"code": "FRA", "city": "Frankfurt", "country": "DE"},
  {"code": "MAD", "city": "Madrid", "country": "ES"},
  {"code": "SFO", "city": "San Francisco CA", "country": "US"},
  {"code": "GRU", "city": "São Paulo", "country": "BR"},
  {"code": "DME", "city": "Moscow", "country": "RU"}
]
```

按照顺序，我们来创建 api/AirCheapAPI.js 文件吧。它包含一个名为 fetchAirports 的函数，用于从远程 json 文件中加载机场信息，并调用 Action 创建器来分发成功或错误 Action。代码清单 6-17 展示了该文件的第一个草稿。

代码清单 6-17: api/AirCheapAPI.js 文件的第一次尝试

```
import 'whatwg-fetch';

let AirCheapAPI = {
  fetchAirports() {
    fetch('airports.json')
      .then((response) => response.json())
      .then((responseData) => {
        // Call the AirportActionCreators success action with the parsed data
      })
      .catch((error) => {
        // Call the AirportActionCreators error action with the error object
      });
  }
};

export default AirCheapAPI;
```

#### 注意:

像之前的示例一样，你会使用原生的 fetch 函数来加载 json 文件，并导入了 whatwg-fetch 这个 npm 模块为旧版浏览器提供 fetch 支持。别忘了使用 `npm install --save`



whatwg-fetch来安装它。

API 模块被调用后, 就会去获取远程数据, 并通过 Action 创建器来将分发成功或失败 Action。你现在还没有 AirportActionCreators, 但可以假设它包含 fetchAirportsSuccess 和 fetchAirportsError 函数, 这样你就可以完成 AirCheapAPI 的实现, 如代码清单 6-18 所示。

代码清单 6-18: 完整的 api/AirCheapAPI.js, 分发了成功或失败 Action

```
import 'whatwg-fetch';
import AirportActionCreators from '../actions/AirportActionCreators';

let AirCheapAPI = {
  fetchAirports() {
    fetch('airports.json')
      .then((response) => {
        return response.json();
      })
      .then((responseData) => {
        // Call the AirportActionCreators success action with the parsed data
        AirportActionCreators.fetchAirportsSuccess(responseData);
      })
      .catch((error) => {
        // Call the AirportActionCreators error action with the error object
        AirportActionCreators.fetchAirportsError(error);
      });
  }
};

export default AirCheapAPI;
```

然后是 AirportActionCreators, 请记住 Action 就像消息, 会被分发到所有 Store: 它们仅用来传达应用程序发生了什么。Action 中并不包含业务逻辑或者计算。理解了这一点之后, 开发 Action 创建器模块就非常简单了。代码清单 6-19 展示了 AirportActionCreators 文件。

代码清单 6-19: actions/AirportActionCreators.js 文件

```
import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import AirCheapAPI from '../api/AirCheapAPI';

let AirportActionCreators = {
  fetchAirports() {
    AirCheapAPI.fetchAirports();
    AppDispatcher.dispatch({
      type: constants.FETCH_AIRPORTS,
    });
  }
};
```



```

    },

    fetchAirportsSuccess(response) {
      AppDispatcher.dispatch({
        type: constants.FETCH_AIRPORTS_SUCCESS,
        payload: {response}
      });
    },

    fetchAirportsError(error) {
      AppDispatcher.dispatch({
        type: constants.FETCH_AIRPORTS_ERROR,
        payload: {error}
      });
    }
  };

  export default AirportActionCreators;

```

### 6.5.3 AirportStore

机场Store可以响应所有可能被分发的Action。它可以响应fetchAirports，设置一个变量来指示当前正在进行加载。它可以响应fetchAirportsError，设置一个变量，用来呈现恰当的错误信息，当然很明显，它也可响应fetchAirportsSuccess，将获取到的机场代码清单设置到其内部state中。

下面从最简单的事项开始：创建 AirportStore.js，使其继承 ReduceStore。它的 state 将包含机场代码清单，首先是一个空数组，然后在响应 fetchAirportsSuccess Action 时会进行填充。代码清单 6-20 展示了完整的源代码。

代码清单 6-20: actions/AirportStore.js 源文件

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {ReduceStore} from 'flux/utils';

class AirportStore extends ReduceStore {
  getInitialState() {
    return [];
  }

  reduce(state, action){
    switch (action.type) {

      case constants.FETCH_AIRPORTS_SUCCESS:
        return action.payload.response;

      default:

```

```

    return state;
  }
}
export default new AirportStore(AppDispatcher);

```

#### 6.5.4 应用组件

接下来实现 AirCheap 应用程序的界面。用户通过输入两个文本域(出发地和目的地)与应用程序交互, 为用户操作变得简单, 你会实现一个自动建议功能, 根据用户的输入给出建议, 如图 6-9 所示。

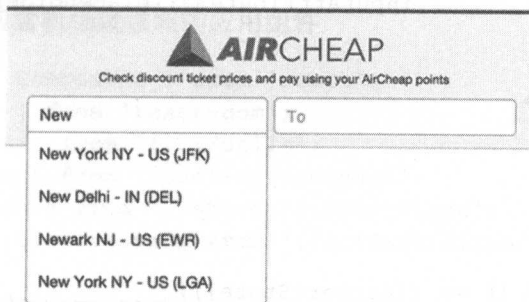


图 6-9 组件及其自动建议功能

有许多提供自动建议功能的库(在 [npmjs.com](http://npmjs.com) 搜索一下就能发现)。在本例中, 你将使用 `react-auto-suggest`, 所以别忘了使用 NPM 来安装它(`npm install -save react-autosuggest`)。

首先, 你会创建应用组件的基本结构。它使用 Flux 工具的容器(用来侦听 Store 变化并将 Store state 映射到组件私有 state), 并会在其生命周期方法 `componentDidMount` 中调用 `AirportActionCreator` 来触发机场的异步加载行为。代码清单 6-21 展示了 `App.js` 的基本结构:

代码清单 6-21: 基本的 `app.js` 组件结构

```

import React, { Component } from 'react';
import { render } from 'react-dom';
import { Container } from 'flux/utils';
import Autosuggest from 'react-autosuggest';
import AirportStore from '../stores/AirportStore';
import AirportActionCreators from '../actions/AirportActionCreators';

class App extends Component {
  componentDidMount() {
    AirportActionCreators.fetchAirports();
  }

  render() {
    return (

```

```

<div>
  <header>
    <div className="header-brand">
      
      <p>Check discount ticket prices and pay using your AirCheap
        points</p>
    </div>
    <div className="header-route">
      <Autosuggest id='origin'
        inputAttributes={{placeholder:'From'}} />

      <Autosuggest id='destination'
        inputAttributes={{placeholder:'To'}} />
    </div>
  </header>
</div>
);
}
}

App.getStores = () => ([AirportStore]);
App.calculateState = (prevState) => ({
  airports: AirportStore.getState()
});

const AppContainer = Container.create(App);
render(<AppContainer />, document.getElementById('root'));

```

如果你现在运行这个应用程序，`react-auto-suggest` 库会抛出一个错误。它要求你将 `suggestions` 函数作为 `props` 传递。用户每次更改了文本框中的输入值后，都会调用这个函数，并会返回一组建议以供展示。代码清单 6-22 展示了这个函数。

#### 代码清单 6-22: `getSuggestions` 函数

```

getSuggestions(input, callback) {
  const escapedInput = input.trim().toLowerCase();
  const airportMatchRegex = new RegExp('\\b'+ escapedInput, 'i');
  const suggestions = this.state.airports
    .filter(airport => airportMatchRegex.test(airport.city))
    .sort((airport1, airport2) => {
      airport1.city.toLowerCase().indexOf(escapedInput) -
      airport2.city.toLowerCase().indexOf(escapedInput)
    })
    .slice(0, 7)
    .map(airport => `${airport.city} - ${airport.country}
      (${airport.code})`);
  callback(null, suggestions);
}

```

该函数接受两个参数：用户输入的文本和包含建议的回调函数。

在函数的开头几行，你清理用户输入，删除多余的空格并将其整体转换为小写状态。在后续代码中，你创建了一个包含用户输入的正则表达式。这个正则表达式随后会被用来筛选机场(基于城市名)。

除了筛选机场外，你还做了另外三项转换。你对机场代码清单进行重新排序，若用户输入在城市名称中出现得越靠前，那么该城市也就越靠前；你将结果数量限制为最多 7 个，你将输出映射为指定的格式：“城市名称 - 国家缩写(机场代码)”。

代码清单 6-23 展示了更新后的应用组件代码，它包含了以 props 形式传递给自动建议组件的 suggestions 函数。代码清单 6-24 则展示了应用程序样式所对应的 CSS 文件。

代码清单 6-23：包含机场建议域的应用组件

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import { Container } from 'flux/utils';
import Autosuggest from 'react-autosuggest';
import AirportStore from '../stores/AirportStore';
import AirportActionCreators from '../actions/AirportActionCreators';

class App extends Component {
  getSuggestions(input, callback) {
    const escapedInput = input.trim().toLowerCase();
    const airportMatchRegex = new RegExp('\\b' + escapedInput, 'i');
    const suggestions = this.state.airports
      .filter(airport => airportMatchRegex.test(airport.city))
      .sort((airport1, airport2) => {
        return airport1.city.toLowerCase().indexOf(escapedInput) -
          airport2.city.toLowerCase().indexOf(escapedInput)
      })
      .slice(0, 7)
      .map(airport => `${airport.city} - ${airport.country}
        (${airport.code})`);
    callback(null, suggestions);
  }

  componentDidMount() {
    AirportActionCreators.fetchAirports();
  }

  render() {
    return (
      <div>
        <header>
          <div className="header-brand">
            
            <p>Check discount ticket prices and pay using your AirCheap
              points</p>
          </div>
        </header>
      </div>
    );
  }
}
```

```

    </div>
    <div className="header-route">
      <Autosuggest id='origin'
        suggestions={this.getSuggestions.bind(this)}
        inputAttributes={{placeholder:'From'}} />

      <Autosuggest id='destination'
        suggestions={this.getSuggestions.bind(this)}
        inputAttributes={{placeholder:'To'}} />
    </div>

  </header>
</div>
);
}
}

App.getStores = () => ([AirportStore]);
App.calculateState = (prevState) => ({
  airports: AirportStore.getState()
});

```

```
const AppContainer = Container.create(App);
```

```
render(<AppContainer />, document.getElementById('root'));
```

#### 代码清单 6-24: AirCheap 应用程序样式表

```

* {
  box-sizing: border-box;
}
body {
  margin: 0;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}
header {
  padding-top: 10px;
  border-bottom: 1px solid #ccc;
  border-top: 4px solid #08516E;
  height: 115px;
  background-color: #f6f6f6;
}
p {
  margin: 0;
  font-size: 10px;
}
.header-brand {
  text-align: center;
}
.header-route {

```

```

margin-top: 10px;
margin-left: calc(50% - 205px)
}
.react-autosuggest{
  position: relative;
  float: left;
  margin-right: 5px;
}
.react-autosuggest input {
  width: 200px;
  height: 30px;
  padding: 14px 10px;
  font-size: 13px;
  border: 1px solid #aaaaaa;
  border-radius: 4px;
}
.react-autosuggest input[aria-expanded="true"]{
  border-bottom-left-radius: 0;
  border-bottom-right-radius: 0;
}
.react-autosuggest input: focus{
  outline: none;
}
.react-autosuggest__suggestions{
  position: absolute;
  top: 29px;
  width: 200px;
  margin: 0;
  padding: 0;
  list-style-type: none;
  border: 1px solid #aaaaaa;
  background-color: #fff;
  font-size: 13px;
  border-bottom-left-radius: 4px;
  border-bottom-right-radius: 4px;
  z-index: 2;
}
.react-autosuggest__suggestions-section-suggestions{
  margin: 0;
  padding: 0;
  list-style-type: none;
}
.react-autosuggest__suggestion{
  cursor: pointer;
  padding: 10px 10px;
}
.react-autosuggest__suggestion--focused{
  background-color: #ddd;
}

```



```

.ticket{
  padding: 20px 10px;
  background-color: #fafafa;
  margin: 5px;
  border: 1px solid #e5e5df;
  border-radius: 3px;
  box-shadow: 0 1px 0 rgba(0, 0, 0, 0.25);
}
.ticket span {
  display: inline-block;
}
.ticket-company{
  font-weight: bold;
  font-style: italic;
  width: 13%;
}
.ticket-location{
  text-align: center;
  width: 29%;
}
.ticket-separator{
  text-align: center;
  width: 6%;
}
.ticket-connection{
  text-align: center;
  width: 10%;
}
.ticket-points{
  width: 13%;
  text-align: right;
}

```

如果你现在测试这个应用程序，就可以看到自动建议已经生效了：只要输入几个字母即可。但应用程序还没有完成。在选择出发地和目的地之后，什么都没有发生，下面接着实现机票的加载。

### 6.5.5 完成 AirCheap 应用程序：加载机票

应用组件挂载后，你会立即异步获取机场数据，但还有一个获取操作没有完成：当用户选择了出发地和目的地之后，你需要获取实际的机票代码清单。

这个过程和获取机场非常相似。你把获取实际数据的代码添加到 API 助手模块。你创建 Action 创建器来发送数据获取步骤(初始化加载过程、加载数据成功、加载失败)的信号，并编写一个新的 Store 将加载好的机票保存到其内部 state 中。应用组件会连接到该 Store，并展示已加载的机票数据。

## 1. API 助手

为简单起见,我们不使用真实 API 来返回航班和机票代码清单,而是从一个静态 json 文件(flights.json)来加载它们。很明显,无论用户选择的机场是什么,加载到的机票总是一样的,但由于你所关注的是学习 Flux 架构,这样的功能也就足够了。代码清单 6-25 展示了 flights.json 文件,包含从 São Paulo (GRU)到 New York (JFK)的剩余航班机票。

代码清单 6-25: flights.json 文件

```
[
  {
    "id": "fc704c16fd79",
    "company": "US Airlines",
    "points": 25000,
    "duration": 590,
    "segment": [
      {
        "duration": 590,
        "departureTime": "2016-10-10T21:30-03:00",
        "arrivalTime": "2016-10-11T06:20-04:00",
        "origin": "GRU",
        "destination": "JFK"
      }
    ]
  },
  {
    "id": "3fe21e46fd78",
    "company": "Dalta",
    "points": 20000,
    "duration": 862,
    "segment": [
      {
        "duration": 635,
        "departureTime": "2016-10-16T20:25-03:00",
        "arrivalTime": "2016-10-17T06:00-04:00",
        "origin": "GRU",
        "destination": "YYZ",
        "connectionDuration": 125
      },
      {
        "duration": 102,
        "departureTime": "2016-10-17T08:05-04:00",
        "arrivalTime": "2016-10-17T09:47-04:00",
        "origin": "YYZ",
        "destination": "JFK"
      }
    ]
  }
]
```

```

    "id": "8bf2b3d7be09",
    "company": "Aviana",
    "points": 17000,
    "duration": 1050,
    "segment": [
      {
        "duration": 515,
        "departureTime": "2016-10-10T21:25-03:00",
        "arrivalTime": "2016-10-11T05:00-04:00",
        "origin": "GRU",
        "destination": "MIA",
        "connectionDuration": 145
      },
      {
        "duration": 192,
        "departureTime": "2016-10-11T07:25-04:00",
        "arrivalTime": "2016-10-11T10:37-04:00",
        "origin": "MIA",
        "destination": "YYZ",
        "connectionDuration": 98
      },
      {
        "duration": 100,
        "departureTime": "2016-10-11T12:15-04:00",
        "arrivalTime": "2016-10-11T13:55-04:00",
        "origin": "YYZ",
        "destination": "JFK"
      }
    ]
  }
]
]

```

接下来编辑 `AirCheapApi.js` 模块，添加方法来获取这个 `json` 文件，并分发对应的 `Action`。正如你在创建 `AirCheapAPI` 文件时做的那样，你再一次假设稍后会在 `AirportActionCreators` 中实现几个方法(`fetchTicketsSuccess` 和 `fetchTicketsError`)。代码清单 6-26 展示了更新后的文件。

代码清单 6-26: 更新后的 `AirCheapAPI.js` 能够获取机票

```

import 'whatwg-fetch';
import AirportActionCreators from '../actions/AirportActionCreators';

let AirCheapAPI = {
  fetchAirports() {
    fetch('airports.json')
      .then((response) => response.json())
      .then((responseData) => {
        AirportActionCreators.fetchAirportsSuccess(responseData);
      })
  }
}

```

```

    .catch((error) =>{
      AirportActionCreators.fetchAirportsError(error);
    });
  },

  fetchTickets(origin, destination) {
    fetch('flights.json')
      .then((response) =>response.json())
      .then((responseData) =>{
        AirportActionCreators.fetchTicketsSuccess(responseData);
      })
      .catch((error) =>{
        AirportActionCreators.fetchTicketsError(error);
      });
  };
};

export default AirCheapAPI;

```

## 2. Action 创建器

下面接着编辑 AirportActionCreators.js 文件。当然，你需要为机票获取操作添加三个必要的 Action 创建器，但首先我们来实现另一个 Action 创建器：chooseAirport。

你在界面上为用户提供了两个自动建议域，来帮助用户选择出发地和目的地，但当用户选择了机场后却什么都没有发生。而 chooseAirport Action 创建器正是用于此目的：每次选中机场(出发地或目的地)时都会触发它。代码清单6-27展示了更新后的 AirportActionCreators。

代码清单 6-27：添加获取机票的 Action 创建器

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import AirCheapAPI from '../api/AirCheapApi';

let AirportActionCreators = {

  fetchAirports() {...},
  fetchAirportsSuccess(response) {...},
  fetchAirportsError(error) {...},

  chooseAirport(target, code) {
    AppDispatcher.dispatch({
      type: constants.CHOOSE_AIRPORT,
      target,
      code
    });
  },
};

```

```

fetchTickets() {
  AirCheapAPI.fetchTickets();
  AppDispatcher.dispatch({
    type: constants.FETCH_TICKETS,
  });
},

fetchTicketsSuccess(response) {
  AppDispatcher.dispatch({
    type: constants.FETCH_TICKETS_SUCCESS,
    payload: {response}
  });
},

fetchTicketsError(error) {
  AppDispatcher.dispatch({
    type: constants.FETCH_TICKETS_ERROR,
    payload: {error}
  });
}
};

```

**export default** AirportActionCreators;

### 3. Store

接下来你会创建两个 Store。第一个 Store 是 RouteStore，保存用户选中的出发地和目的地机场。第二个 Store 是 TicketStore，当两个机场都选好并加载了航班机票数据后，将其保存起来。

下面从 RouteStore 开始着手。它继承自 MapStore，允许保存多个键值对。键会有两种可能：origin 和 desination。而且 Store 还会响应 CHOOSE\_AIRPORT Action 来选取这两个键中的一个，将其对应的值更新为机场代码。代码清单 6-28 展示了完整的源代码。

代码清单 6-28: stores/RouteStore.js 文件的完整源代码

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {MapStore} from 'flux/utils';

class RouteStore extends MapStore {
  reduce(state, action){
    switch (action.type) {
      case constants.CHOOSE_AIRPORT:
        // action.target can be either "origin" or "destination"
        // action.code contains the selected airport code
        return state.set(action.target,action.code);
      default:
        return state;
    }
  }
}

```

```

    }
  }
  export default new RouteStore(AppDispatcher);

```

TicketStore 和 AirportStore 非常相似。它继承自 ReduceStore, 当 FETCH\_TICKETS\_SUCCESS Action 被分发时, 它就会更新自己的内部 state。代码清单 6-29 展示了完整的源代码。

代码清单 6-29: stores/TicketStore.js 文件的源代码

```

import AppDispatcher from '../AppDispatcher';
import AirportActions from '../actions/AirportActionCreators';
import constants from '../constants';
import RouteStore from './RouteStore';
import {ReduceStore} from 'flux/utils';

class TicketStore extends ReduceStore {
  getInitialState() {
    return [];
  }
  reduce(state, action) {
    switch (action.type) {
      case constants.FETCH_TICKETS:
        return [];
      case constants.FETCH_TICKETS_SUCCESS:
        return action.payload.response;
      default:
        return state;
    }
  }
}

export default new TicketStore(AppDispatcher);

```

注意, TicketStore 还响应了 FETCH\_TICKETS Action, 并将其 state 设置为空数组。这样, 每次尝试获取不同的机票时, 界面就能立即清空之前可能存在的任何机票信息了。

#### 4. 界面组件

下面开始着手处理界面吧, 首先创建一个新组件: TicketItem.js。它会以 props 的形式接收组件信息, 并展示一个机票行。代码清单 6-30 展示了组件的代码。

代码清单 6-30: components/TicketItem.js 组件

```

import React, { Component, PropTypes } from 'react';

// Default data configuration
const dateConfig = {
  weekday: "short",
  year: "numeric",

```



```

    month: "short",
    day: "numeric",
    hour: "2-digit",
    minute: "2-digit"
  });

class TicketItem extends Component {
  render() {
    let {ticket} = this.props;
    let departureTime = new Date(ticket.segment[0].departureTime)
      .toLocaleDateString("en-US", dateConfig);
    let arrivalTime = new Date(ticket.segment[ticket.segment.length-1]
      .arrivalTime).toLocaleDateString("en-US", dateConfig);

    let stops;
    if(ticket.segment.length === 2){
      stops = '1 stop';
    } else if(ticket.segment.length-1 > 1) {
      stops = ticket.segment.length-1 + ' stops';
    }

    return(
      <div className='ticket'>
        <span className="ticket-company">{ticket.company}</span>
        <span className="ticket-location">
          <strong>{ticket.segment[0].origin}</strong>{' '}
          <small>{departureTime}</small>
        </span>
        <span className="ticket-separator">

        </span>
        <span className="ticket-location">
          <strong>{ticket.segment[ticket.segment.length-1].destination}</strong>{' '}
          <small>{arrivalTime}</small>
        </span>
        <span className="ticket-connection">
          {stops}
        </span>
        <span className="ticket-points">
          <button>{ticket.points} points</button>
        </span>
      </div>
    );
  }
}

TicketItem.propTypes = {
  ticket: PropTypes.shape({
    id: PropTypes.string,

```

```

    company: PropTypes.string,
    points: PropTypes.number,
    duration: PropTypes.number,
    segment: PropTypes.array
  }},
  App extends Component {
    // ...
  };

export default TicketItem;

```

按照顺序，下面更新主应用组件。你需要完成如下几件事项：

- 让组件侦听新 Store(RouteStore 和 TicketStore)的更新，并用这两个 Store 的状态计算其 state。可通过编辑静态方法 `getStores` 和 `calculateState` 来完成这一任务：

```

App.getStores= () => ([AirportStore,RouteStore,TicketStore]);
App.calculateState= (prevState) => ({
  airports:AirportStore.getState(),
  origin:RouteStore.get('origin'),
  destination:RouteStore.get('destination'),
  tickets:TicketStore.getState()
});

```

- 当用户选择了出发和目的机场之后，调用 `chooseAirport` Action 创建器。方法是向自动建议的 `onSuggestionSelected` props 传递一个回调函数。你可以有两个不同的回调(一个用于出发域，另一个用于目的域)，但使用 JavaScript 的 `bind` 函数，你可以只绑定一个回调函数，并为每个域传递不同的参数：

```

<Autosuggest id='origin'
  suggestions={this.getSuggestions.bind(this)}
  onSuggestionSelected={this.handleSelect.bind(this,'origin')}
  value={this.state.origin}
  inputAttributes={{placeholder:'From'}} />
<Autosuggest id='destination'
  suggestions={this.getSuggestions.bind(this)}
  onSuggestionSelected={this.handleSelect.bind(this,'destination')}
  value={this.state.destination}
  inputAttributes={{placeholder:'To'}} />

```

`handleSelect` 函数使用了一个正则表达式从字符串中提取机场代码，并调用 `chooseAirport` Action 创建器：

```

handleSelect(target, suggestion, event){
  const airportCodeRegex =/\(([^\)]+)\)/;
  let airportCode =airportCodeRegex.exec(suggestion)[1];
  AirportActionCreators.chooseAirport(target, airportCode);
}

```

- 当用户选中了出发和目的机场之后，调用 `fetchTickets` Action 创建器。你可在生命周期方法 `componentWillUpdate` 中完成这一任务；用户每次选择机场时，都调用了 Action 创建器，这会导致 RouteStore 分发一个变化事件，应用组件就会被更新。

你在调用Action创建器之前检查两件事情：出发地和目的地是否都已选中，以及两者中是否有一个自上次更新后发生了变化(这样你就不会重复获取机票了)：

```
componentWillUpdate(nextProps, nextState) {
  let originAndDestinationSelected =
    nextState.origin && nextState.destination;
  let selectionHasChangedSinceLastUpdate =
    nextState.origin !== this.state.origin ||
    nextState.destination !== this.state.destination;
  if (originAndDestinationSelected &&
    selectionHasChangedSinceLastUpdate) {
    AirportActionCreators.fetchTickets(
      nextState.origin, nextState.destination);
  }
}
```

- 最后，导入并实现你刚才创建的 TicketItem 组件，展示加载好的机票：

```
render() {
  let ticketList = this.state.tickets.map((ticket)=>(
    <TicketItem key={ticket.id} ticket={ticket} />
  ));
  return (
    <div>
      <header>
        <div className="header-brand">...</div>
        <div className="header-route">
          <Autosuggest id='origin' ... />
          <Autosuggest id='destination' ... />
        </div>
      </header>
      <div>
        {ticketList}
      </div>
    </div>
  );
}
```

代码清单 6-31 展示了更新后的应用组件，包含了以上提及的所有改动。

#### 代码清单 6-31：更新后的应用组件

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import ReactDOM from 'react-dom';
import { Container } from 'flux/utils';
import Autosuggest from 'react-autosuggest';
import AirportStore from '../stores/AirportStore';
import RouteStore from '../stores/RouteStore';
```

```

import TicketStore from './stores/TicketStore';
import TicketItem from './components/TicketItem';
import AirportActionCreators from './actions/AirportActionCreators';

class App extends Component {
  getSuggestions(input, callback) {
    const escapedInput = input.trim().toLowerCase();
    const airportMatchRegex = new RegExp('\\b' + escapedInput, 'i');
    const suggestions = this.state.airports
      .filter(airport => airportMatchRegex.test(airport.city))
      .sort((airport1, airport2) => {
        return airport1.city.toLowerCase().indexOf(escapedInput) -
          airport2.city.toLowerCase().indexOf(escapedInput)
      })
      .slice(0, 7)
      .map(airport => `${airport.city} - ${airport.country}
        (${airport.code})`);
    callback(null, suggestions);
  }

  handleSelect(target, suggestion, event){
    const airportCodeRegex = /\([([^\)]+)\)/;
    let airportCode = airportCodeRegex.exec(suggestion)[1];
    AirportActionCreators.chooseAirport(target, airportCode);
  }

  componentDidMount() {
    AirportActionCreators.fetchAirports();
  }

  componentWillUpdate(nextProps, nextState){
    let originAndDestinationSelected = nextState.origin &&
      nextState.destination;

    let selectionHasChangedSinceLastUpdate = nextState.origin !==
      this.state.origin || nextState.destination !== this.state.
        destination;

    if(originAndDestinationSelected && selectionHasChangedSinceLastUpdate){
      AirportActionCreators.fetchTickets(nextState.origin,
        nextState.destination);
    }
  }

  render() {
    let ticketList = this.state.tickets.map((ticket)=>(
      <TicketItem key={ticket.id} ticket={ticket} />
    ));
    return (
      <div>
        <header>
          <div className="header-brand">

```

```

    
    <p>Check discount ticket prices and pay using your AirCheap
      points</p>
  </div>
  <div className="header-route">
    <Autosuggest id='origin'
      suggestions={this.getSuggestions.bind(this)}
      onSuggestionSelected={this.handleSelect.bind(
        this, 'origin')}
      value={this.state.origin}
      inputAttributes={{placeholder: 'From'}} />
    <Autosuggest id='destination'
      suggestions={this.getSuggestions.bind(this)}
      onSuggestionSelected={this.handleSelect.bind(
        this, 'destination')}
      value={this.state.destination}
      inputAttributes={{placeholder: 'To'}} />
  </div>
</header>
<div>
  {ticketList}
</div>
</div>
);
}
}

App.getStores = () => ([AirportStore, RouteStore, TicketStore]);
App.calculateState = (prevState) => ({
  airports: AirportStore.getState(),
  origin: RouteStore.get('origin'),
  destination: RouteStore.get('destination'),
  tickets: TicketStore.getState()
});

const AppContainer = Container.create(App);

render(<AppContainer />, document.getElementById('root'));
```

如果你现在进行测试，应用程序应该能够正常工作，而且在选择了出发地和目的地之后加载机票了。

## 6.6 改进异步获取数据的实现

你已经知道，在 Flux 中进行异步 API 通信的最佳方式是将所有 API 相关的代码封装到一个 API 助手模块中。你会通过一个 Action 来调用该 API 助手模块，API 助手模块异

步加载的所有远程数据也都通过 Action 进入系统。这是一个优雅的解决方案，它遵循了 Flux 的单向数据流原则，并将异步代码和系统的其他功能分离。然而我们还可以进一步改进这一模型，移除一些样板代码，再将 API 助手模块和 Action 创建器解耦。为达成此目标，你将在 AppDispatcher 中增加一个新方法：dispatchAsync。

## AppDispatcher 的 dispatchAsync

Flux 的 Dispatcher 只包含几个公共方法，一般使用最多的就是 dispatch。你已经知道，dispatch 方法可用来将 Action 分发到所有已注册的 Store。

正如你在之前关于异步 API 的主题(以及 AirCheap 示例应用程序)中所看到的，每个异步操作都对应三个 Action(异步操作请求、成功和失败)。通用的 dispatchAsync 方法的参数是一个 Promise 以及用来表示异步操作所有步骤(请求、成功、失败)的常量，它会根据 Promise 的解析状态自动分发它们。

代码清单 6-32 展示了更新后的 AppDispatcher，包含了 dispatchAsync 方法的实现。注意使用了 Babel polyfill 来确保 Object.assign 能在旧版浏览器中工作(请务必使用 npm install --save babel-polyfill 来安装它)。

代码清单 6-32: 包含 dispatcherAsync 的 AppDispatcher

```
import {Dispatcher} from 'flux';
import 'babel-polyfill';

class AppDispatcher extends Dispatcher{
  dispatch(action = {}) {
    console.log("Dispatched", action.type);
    super.dispatch(action);
  }

  /**
   * Dispatches three actions for an async operation represented by promise.
   */
  dispatchAsync(promise, types, payload){
    const{ request, success, failure }= types;
    this.dispatch({type: request,payload:Object.assign({}, payload) });
    promise.then(
      response =>this.dispatch({
        type: success,
        payload:Object.assign({}, payload,{ response })
      }),
      error =>this.dispatch({
        type: failure,
        payload:Object.assign({}, payload,{ error })
      })
    );
  }
}
```



```

    }
  },
  export default new AppDispatcher();

```

使用这个方法，你可在 Action 创建器中少输入许多代码，因为你为每个异步操作只创建一个方法，而不再是三个。代码清单 6-33 展示了更新后的 AirportActionCreators 文件作为示例。

代码清单 6-33: 更新后的 AirportActionCreators.js(缩短了 50%)

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import AirCheapAPI from '../api/AirCheapApi';

let AirportActionCreators = {

  fetchAirports(origin, destination) {
    AppDispatcher.dispatchAsync(AirCheapAPI.fetchAirports(), {
      request: constants.FETCH_AIRPORTS,
      success: constants.FETCH_AIRPORTS_SUCCESS,
      failure: constants.FETCH_AIRPORTS_ERROR
    });
  },

  chooseAirport(target, code) {
    AppDispatcher.dispatch({
      type: constants.CHOOSE_AIRPORT,
      target: target,
      code: code
    });
  },

  fetchTickets(origin, destination) {
    AppDispatcher.dispatchAsync(AirCheapAPI.fetchTickets(
      origin, destination), {
      request: constants.FETCH_TICKETS,
      success: constants.FETCH_TICKETS_SUCCESS,
      failure: constants.FETCH_TICKETS_ERROR
    });
  }
};

export default AirportActionCreators;

```

在 API 助手模块中，你不仅减少了样板代码，还将其与 Action 创建器解耦，因为 API 助手不再需要直接调用成功或失败所对应的方法了。它所需要做的仅是返回一个 Promise。代码清单 6-34 展示了更新后的 AirCheapApi.js 文件，返回了由 fetch 操作创建

的 Promise，并且链式执行了 JSON 解析操作。

代码清单 6-34：更新后的 AirCheapApi.js 文件

```
import 'whatwg-fetch';

let AirCheapAPI = {
  fetchAirports() {
    return fetch('airports.json')
      .then((response) => response.json());
  },

  fetchTickets(origin, destination) {
    return fetch('flights.json')
      .then((response) => response.json());
  }
};

export default AirCheapAPI;
```

## 6.7 看板应用：迁移到 Flux 架构

你从本书开篇就一直在开发看板应用，并且在每一章中都会为它增加新功能。然而本章会有些变化。想必你已经了解到，Flux 并非为 React 项目带来新功能的必需品。Flux 是一种应用架构，它能让应用中的数据变化更易于理解。在将你的看板应用转换到 Flux 架构时，你并不会增加功能；你会让它更容易预测和理解(从这个意义上讲，当然也有助于在将来添加新功能)。

### 6.7.1 重构：创建 Flux 基本结构并迁移文件

在开始之前，务必在项目中使用 npm 安装 Flux：npm install --save flux。接下来，为 Flux 文件创建一个文件夹，并将你的所有组件(App.js 文件除外)移到一个 components 文件夹中。常量和工具文件依然存放在 app 根目录下。图 6-10 展示了新的文件夹结构。

#### 1. 修复 import

很明显，你需要更新组件中的 import 语句来反应新的文件夹结构。幸运的是，这些导入都是相对的，所以你不需要修改所有组件。受影响的组件只有：

- App.js(你需要纠正导入的所有组件的路径)
- KanbanBoardContainer.js(你只需要更新对工具的 import)
- Card.js 和 List.js(你需要修复对 onstants.js 模块的 import)

代码清单 6-35 到代码清单 6-38 展示了上述所有文件更新 import 之后的代码。

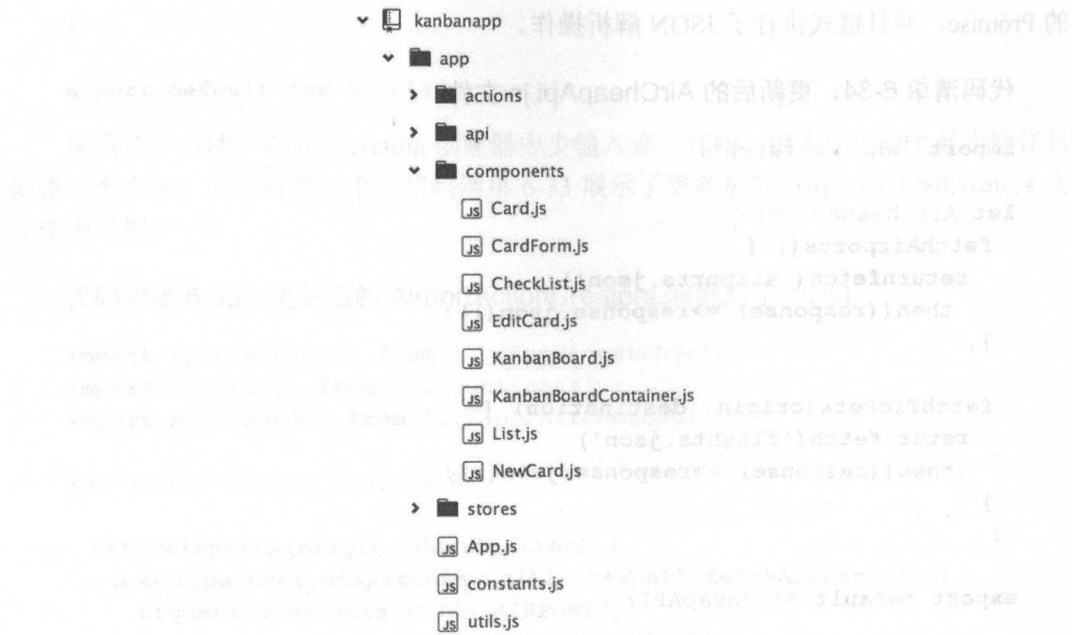


图 6-10 看板应用的新文件夹结构

代码清单 6-35: 在 App.js 中更新的 import

```
import React from 'react';
import { render } from 'react-dom';
import { Router, Route } from 'react-router';
import createBrowserHistory from 'history/lib/createBrowserHistory';
import KanbanBoardContainer from '../components/KanbanBoardContainer';
import KanbanBoard from '../components/KanbanBoard';
import EditCard from '../components/EditCard';
import NewCard from '../components/NewCard';

render(...);
```

代码清单 6-36: 在 KanbanBoardContainer 组件中更新了对工具的 import

```
import React, { Component } from 'react';
import update from 'react-addons-update';
import KanbanBoard from '../KanbanBoard';

import { throttle } from '../utils';

import 'babel-polyfill'
import 'whatwg-fetch';

const API_URL = 'http://kanbanapi.pro-react.com';
const API_HEADERS = {...};

class KanbanBoardContainer extends Component {
```

```

constructor(){...}
componentDidMount(){...}
addCard(card){...}
updateCard(card){...}
updateCardStatus(cardId, listId){...}
updateCardPosition (cardId , afterId) {...}
persistCardDrag(cardId, status){...}
addTask(cardId, taskName){...}
deleteTask(cardId, taskId, taskIndex){...}
toggleTask(cardId, taskId, taskIndex){...}
render() {...}
}

export default KanbanBoardContainer;

```

#### 代码清单 6-37: 在卡片组件中更新的 import

```

import React,{ Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from '../constants';
import CheckList from './CheckList';
import {Link}from'react-router';

let titlePropType = (props, propName, componentName) =>{...}
const cardDragSpec ={...}
const cardDropSpec ={...}
let collectDrag = (connect, monitor) =>{...}
let collectDrop = (connect, monitor) =>{...}

class Card extends Component {...}
Card.propTypes={...}

const dragHighOrderCard = ...
const dragDropHighOrderCard = ...

export default dragDropHighOrderCard

```

#### 代码清单 6-38: 在列表组件中更新的 import

```

import React,{ Component, PropTypes } from 'react';
import { DropTarget } from 'react-dnd';
import Card from './Card';
import constants from '../constants';

const listTargetSpec ={...};
function collect(connect, monitor) {...}

class List extends Component {...}

```

```
List.propTypes={...}
```

```
export default DropTarget(CARD, listTargetSpec, collect)(List);
```

## 2. 添加 Flux 基本文件

Flux 就是 Action、Store 和一个 Dispatcher(再加上一个 API 助手来处理 API 异步请求)。为此, 在项目中添加一些新文件:

- 一个 AppDispatcher.js
- 一个 Store: CardStore.js, 位于 stores 文件夹中
- 两个 Action: CardActionCreators.js 和 TaskActionCreators.js, 位于 actions 文件夹中
- 最后, 一个助手模块 KanbanApi.js, 位于 api 文件夹中

你将在 Flux 基础 Dispatcher 中扩展一个 DispatchAsync 方法(你之前已经使用过它了)。对于 CardStore.js、CardActionCreators.js、TaskActionCreators.js 和 KanbanApi.js 这几个文件, 你会先为它们创建最基本的结构, 并在后面的几节中对其进行增强。代码清单 6-39 到代码清单 6-43 展示了这些文件的源代码, 首先是 AppDispatcher。

### 代码清单 6-39: AppDispatcher

```
import {Dispatcher} from 'flux';
import 'babel-polyfill';

class AppDispatcher extends Dispatcher{
  /**
   * Dispatches three actions for an async operation represented by promise.
   */
  dispatchAsync(promise, types, payload){
    const { request, success, failure }= types;
    this.dispatch({type: request,payload:Object.assign({}, payload) });
    promise.then(
      response => this.dispatch({
        type: success,
        payload:Object.assign({}, payload,{ response })
      }),
      error => this.dispatch({
        type: failure,
        payload:Object.assign({}, payload,{ error })
      })
    );
  }
}

export default new AppDispatcher();
```

CardStore 会从 Flux 的 ReduceStore 扩展而来。

代码清单 6-40: stores/CardStore.js 的基本结构

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {ReduceStore} from 'flux/utils';

class CardStore extends ReduceStore {
  getInitialState() {
    return [];
  }

  reduce(state, action) {
    switch (action.type) {
      default:
        return state;
    }
  }
}

export default new CardStore(AppDispatcher);

```

CardActionCreators 和 TaskActionCreators 一开始只是普通的 JavaScript 对象，但你会导入一些将来会用到的模块。

代码清单 6-41: actions/CardActionCreators.js 文件的基本结构

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';

let CardActionCreators = {
};

export default CardActionCreators;

```

代码清单 6-42: actions/TaskActionCreators.js 文件的基本结构

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';

let TaskActionCreators = {
};

export default TaskActionCreators;

```

KanbanApi 同样是普通的 JavaScript 对象，并且也同样使用 import 语句来导入将来会用到的模块。



代码清单 6-43: api/KanbanApi.js 文件的基本结构

```
import 'whatwg-fetch';
import 'babel-polyfill';

let KanbanAPI = {

};

export default KanbanAPI;
```

### 6.7.2 将数据获取操作迁移到 Flux 架构

你的项目结构已经为使用 Flux 做好了准备, 将被迁移到新架构的首批代码是初始数据的获取操作。目前, 所有 API 通信(包括初始数据获取操作)都在 KanbanBoardContainer 内完成, 卡片被保存在组件的 state 中。在 Flux 架构中, KanbanBoardContainer 及其子组件(比如 Card 组件)将只负责触发 Action; API 通信交由 API 助手模块来完成, 卡片则被保存在 CardStore 中。

#### 编辑 KanbanBoardContainer

你现在要处理的仅是初始数据获取操作, 所以你只需要在 KanbanBoardContainer 中做以下改动:

- 导入 CardStore 和 CardActionCreator 模块。
- 让 KanbanBoardContainer 侦听 CardStore 的变化事件, 并将其 state 映射到 CardStore 的 state(你可以手工完成这一操作, 也可以使用 Flux 库中的 Container 高阶函数)。在此过程中, 你还会移除在类构造函数中声明的私有 state。
- ComponentDidMount 生命周期方法不再直接获取数据, 而是调用一个 Action 创建器来分发一个 Action。该 Action 会产生一系列影响。API 助手会获取远程数据, CardStore 会使用新数据来更新自身并分发一个变化事件, 最后 KanbanBoardContainer 的 state 会被更新并因此重新渲染。

代码清单 6-44 展示了更新后的 KanbanBoardContainer 代码。

代码清单 6-44: 更新后的 KanbanBoardContainer

```
import React, { Component } from 'react';
import update from 'react-addons-update';
import KanbanBoard from './KanbanBoard';
import {throttle} from '../utils';

import {Container} from 'flux/utils';
import CardActionCreators from '../actions/CardActionCreators';
import CardStore from '../stores/CardStore';

// Polyfills
```

```

import 'babel-polyfill';
import 'whatwg-fetch';

const API_URL=' http://kanbanapi.pro-react.com'
const API_HEADERS = {...}

class KanbanBoardContainer extends Component {
  constructor(
    super(...arguments);
    this.updateCardStatus=throttle(this.updateCardStatus.bind(this));
    this.updateCardPosition=throttle(this.updateCardPosition.bind(
      this),500);
  )

  componentDidMount(){
    CardActionCreators.fetchCards();
  }

  addCard(card){...}
  updateCard(card){...}
  updateCardStatus(cardId, listId){...}
  updateCardPosition (cardId , afterId){...}
  persistCardDrag(cardId, status){...}
  addTask(cardId, taskName){...}
  deleteTask(cardId, taskId, taskIndex){...}
  toggleTask(cardId, taskId, taskIndex){...}

  render() {...}
}

KanbanBoardContainer.getStores= () => ([CardStore]);
KanbanBoardContainer.calculateState= (prevState) => ({
  cards:CardStore.getState()
});

export default Container.create(KanbanBoardContainer);

```

从 KanbanBoardContainer 中移除处理卡片和任务操作的 8 个方法是最终目标，我们把它列入计划，现在先来处理初始数据获取操作。

### 6.7.3 实现 FetchCards Action、API 方法调用和 Store 回调

到目前为止，你已经完成了两个不同的 Flux 项目(Flux 银行和 AirCheap)，所以应该已经熟悉了流程：你需要定义一个 Action 创建器，当它被调用时会去调用 API 助手，它接受一个 JavaScript Promise，并会在整个过程(获取操作的初始化、成功和失败)中分发不同的 Action。CardStore 会响应成功 Action 来将已加载的卡片填充到其 state 中。

## 1. FetchCards 常量和 Action 创建器

如你所知, 每个 Action 都需要一个常量来作为标识。你已经有了一个常量文件。下面添加三个新的常量: `FETCH_CARDS`、`FETCH_CARDS_SUCCESS` 和 `FETCH_CARDS_ERROR`(如代码清单 6-45 所示)。

代码清单 6-45: 更新后的 constants.js 源代码

```
export default {
  CARD: 'card',
  FETCH_CARDS: 'fetch cards',
  FETCH_CARDS_SUCCESS: 'fetch cards success',
  FETCH_CARDS_ERROR: 'fetch cards error',
};
```

按照顺序, 下面在 `CardActionCreators` 中创建一个 `fetchCards` 方法。你会使用 `AppDispatcher` 的 `DispatchAsync` 方法来让事情更加精简。代码清单 6-46 展示了实现好的方法。

代码清单 6-46: `CardActionCreators` 中的 `fetchCards` 方法实现

```
import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';

let CardActionCreators = {
  fetchCards() {
    AppDispatcher.dispatchAsync(KanbanAPI.fetchCards(), {
      request: constants.FETCH_CARDS,
      success: constants.FETCH_CARDS_SUCCESS,
      failure: constants.FETCH_CARDS_ERROR
    });
  }
};

export default CardActionCreators;
```

注意, 你在代码中假设 `KanbanAPI` 模块拥有一个 `fetchCards` 方法。我们会在下一节中实现它。

## 2. API 方法: `fetchCards`

完成了 `ActionCreators`(以及对应的常量)之后, 下面将注意力转向 `kanbanAPI`。基本上你会把在 `KanbanBoardContainer` 中用到的配置和 `fetch` 方法复制过来, 但在本例中, 你只会简单地返回获取操作的 `Promise`(而不会去操作卡片的 `state`, 现在这一部分属于 `Store` 的责任了), 如代码清单 6-47 所示。

代码清单 6-47: 包含 fetchCards 方法的 KanbanApi 源代码

```
import 'whatwg-fetch';
import 'babel-polyfill';

const API_URL = 'http://kanbanapi.pro-react.com';
const API_HEADERS = {
  'Content-Type': 'application/json',
  Authorization: 'any-string-you-like'
}

let KanbanAPI = {
  fetchCards() {
    return fetch(`${API_URL}/cards`, {headers: API_HEADERS})
      .then((response) => response.json());
  }
};

export default KanbanAPI;
```

### 3. CardStore: 响应 FETCH\_CARDS\_SUCCESS

最后更新 CardStore 中的 reduce 方法, 使其响应 FETCH\_CARD\_SUCCESS 并用已加载的卡片来更新其 state, 如代码清单 6-48 所示。

代码清单 6-48: CardStore 中更新后的 reduce 方法

```
import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {ReduceStore} from 'flux/utils';

class CardStore extends ReduceStore {
  getInitialState() {
    return [];
  }

  reduce(state, action) {
    switch (action.type) {
      case constants.FETCH_CARDS_SUCCESS:
        return action.payload.response;

      default:
        return state;
    }
  }
}

export default new CardStore(AppDispatcher);
```

因为 KanbanBoardContainer 已经在侦听 CardStore 的变化, 你的任务算是完成了。如

果现在进行测试，你应该能正常地查看卡片了。

6.7.4 将所有卡片和任务 Action 迁移到 Flux 架构

在上一节中，你从 KanbanBoardContainer 里移除了获取初始数据的代码，但组件依然包含其他 8 个用来操作卡片和任务的方法。这些方法目前是以 props 的形式向下传递到所有层级的组件，并会被 List、Card 和 Task 组件触发。让我们回顾一下你在阅读本书的过程中创建的这些方法吧：表 6-3 列出了这些方法以及它们的作用。

表 6-3 KanbanBoardContainer 组件中当前的数据操纵方法

方法	描述
addCard	接受一个包含卡片属性的对象作为参数；用来创建一张新卡片
updateCard	接受一个包含更新后的卡片属性的对象；用来更新给定卡片的属性。重构后会接受两个属性：最初的卡片和更改后的卡片
updateCardPosition	接受当前卡片 id 以及将要和当前卡片交换位置的卡片 id。会在拖曳卡片时被调用。用来切换给定卡片的位置
updateCardStatus	接受当前卡片 id 和新的状态 id。会在拖曳卡片时被调用。用来更新卡片状态
persistCardDrag	接受一个包含给定卡片 id 和新卡片状态的对象。会在完成卡片拖曳操作之后被调用。用来将新卡片位置和状态持久化到服务器
addTask	接受一个卡片 id 和一个任务名称；用来为指定的卡片创建一个新任务。重构后，你会传递完整的 Task 对象而非任务名称
deleteTask	接受一个卡片 id、一个任务 id 和任务索引；用来删除任务。重构后，你会传递完整的卡片对象而非 id
toggleTask	接受一个卡片 id、一个任务 id 和任务索引；用来切换任务的 done 属性。重构后，你会传递完整的卡片对象而非 id

所有改动将一并完成。首先你会在 Flux 架构中再造以上所有方法(Action 创建器、KanbanApi 和 CardStore)。然后才会更新 KanbanBoardContainer 及其所有受到影响的子组件(KanbanBoard、List、Card 和 Checklist 组件)。

6.7.5 准备功能迁移

在着手迁移 Action 创建器、API 模块或 Store 之前，先来做些准备工作。你要在常量文件中声明所有必要的常量。代码清单 6-49 展示了更新后的 constants.js 文件。

代码清单 6-49: constants.js 文件中的必要常量声明

```
export default {
```

```

CARD: 'card',

FETCH_CARDS: 'fetch cards',
FETCH_CARDS_SUCCESS: 'fetch cards success',
FETCH_CARDS_ERROR: 'fetch cards error',

CREATE_CARD: 'create card',
CREATE_CARD_SUCCESS: 'create card success',
CREATE_CARD_ERROR: 'create card error',

UPDATE_CARD: 'update card',
UPDATE_CARD_SUCCESS: 'update card success',
UPDATE_CARD_ERROR: 'update card error',

UPDATE_CARD_STATUS: 'update card status',

UPDATE_CARD_POSITION: 'update card position',

PERSIST_CARD_DRAG: 'persist card drag',
PERSIST_CARD_DRAG_SUCCESS: 'persist card drag success',
PERSIST_CARD_DRAG_ERROR: 'persist card drag error',

CREATE_TASK: 'create task',
CREATE_TASK_SUCCESS: 'create task success',
CREATE_TASK_ERROR: 'create task error',

DELETE_TASK: 'delete task',
DELETE_TASK_SUCCESS: 'delete task success',
DELETE_TASK_ERROR: 'delete task error',

TOGGLE_TASK: 'toggle task',
TOGGLE_TASK_SUCCESS: 'toggle task success',
TOGGLE_TASK_ERROR: 'toggle task error'
};

```

### 1. Action 创建器

按照顺序，让我们来实现所有的卡片和任务操作 Action。注意在 CardActionCreators 模块中，导入并使用了 throttle 工具函数。代码清单 6-50 展示了 CardActionCreators.js 文件，代码清单 6-51 则展示了更新后的 TaskActionCreators.js 文件。

#### 代码清单 6-50: CardActionCreators.js

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';
import {throttle} from '../utils';
import CardStore from '../stores/CardStore';

```



```

let CardActionCreators = {

  fetchCards() {
    AppDispatcher.dispatchAsync(KanbanAPI.fetchCards(), {
      request: constants.FETCH_CARDS,
      success: constants.FETCH_CARDS_SUCCESS,
      failure: constants.FETCH_CARDS_ERROR
    });
  },

  addCard(card) {
    AppDispatcher.dispatchAsync(KanbanAPI.addCard(card), {
      request: constants.CREATE_CARD,
      success: constants.CREATE_CARD_SUCCESS,
      failure: constants.CREATE_CARD_ERROR
    }, {card});
  },

  updateCard(card, draftCard) {
    AppDispatcher.dispatchAsync(KanbanAPI.updateCard(card, draftCard), {
      request: constants.UPDATE_CARD,
      success: constants.UPDATE_CARD_SUCCESS,
      failure: constants.UPDATE_CARD_ERROR
    }, {card, draftCard});
  },

  updateCardStatus: throttle((cardId, listId) => {
    AppDispatcher.dispatch({
      type: constants.UPDATE_CARD_STATUS,
      payload: {cardId, listId}
    });
  }, 500),

  updateCardPosition: throttle((cardId, afterId) => {
    AppDispatcher.dispatch({
      type: constants.UPDATE_CARD_POSITION,
      payload: {cardId, afterId}
    });
  }, 500),

  persistCardDrag(cardProps) {
    let card = CardStore.getCard(cardProps.id)
    let cardIndex = CardStore.getCardIndex(cardProps.id)
    AppDispatcher.dispatchAsync(KanbanAPI.persistCardDrag(
      card.id, card.status, cardIndex), {
      request: constants.PERSIST_CARD_DRAG,
      success: constants.PERSIST_CARD_DRAG_SUCCESS,
      failure: constants.PERSIST_CARD_DRAG_ERROR
    }, {cardProps});
  }
};

```

```

    }
  };

  export default CardActionCreators;

```

#### 代码清单 6-51: TaskActionCreators.js

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';

let TaskActionCreators = {
  addTask(cardId, task) {
    AppDispatcher.dispatchAsync(KanbanAPI.addTask(cardId, task), {
      request: constants.CREATE_TASK,
      success: constants.CREATE_TASK_SUCCESS,
      failure: constants.CREATE_TASK_ERROR
    }, {cardId, task});
  },

  deleteTask(cardId, task, taskIndex) {
    AppDispatcher.dispatchAsync(KanbanAPI.deleteTask(cardId, task), {
      request: constants.DELETE_TASK,
      success: constants.DELETE_TASK_SUCCESS,
      failure: constants.DELETE_TASK_ERROR
    }, {cardId, task, taskIndex});
  },

  toggleTask(cardId, task, taskIndex) {
    AppDispatcher.dispatchAsync(KanbanAPI.toggleTask(cardId, task), {
      request: constants.TOGGLE_TASK,
      success: constants.TOGGLE_TASK_SUCCESS,
      failure: constants.TOGGLE_TASK_ERROR
    }, {cardId, task, taskIndex});
  }
};

export default TaskActionCreators;

```

## 2. KanbanApi

接下来,就要开始向 Flux 架构的迁移了,下面更新 KanbanApi 模块,如代码清单 6-52 所示。

#### 代码清单 6-52: 更新后的 KanbanApi

```

import 'whatwg-fetch';
import 'babel-polyfill';

const API_URL = 'http://kanbanapi.pro-react.com';

```

```

const API_HEADERS = {
  'Content-Type': 'application/json',
  Authorization: 'any-string-you-like'
}

let KanbanAPI = {
  fetchCards() {
    return fetch(`${API_URL}/cards`, {headers: API_HEADERS})
      .then((response) => response.json())
  },

  addCard(card) {
    return fetch(`${API_URL}/cards`, {
      method: 'post',
      headers: API_HEADERS,
      body: JSON.stringify(card)
    })
      .then((response) => response.json())
  },

  updateCard(card, draftCard) {
    return fetch(`${API_URL}/cards/${card.id}`, {
      method: 'put',
      headers: API_HEADERS,
      body: JSON.stringify(draftCard)
    })
  },

  persistCardDrag(cardId, status, index) {
    return fetch(`${API_URL}/cards/${cardId}`, {
      method: 'put',
      headers: API_HEADERS,
      body: JSON.stringify({status, row_order_position: index})
    })
  },

  addTask(cardId, task) {
    return fetch(`${API_URL}/cards/${cardId}/tasks`, {
      method: 'post',
      headers: API_HEADERS,
      body: JSON.stringify(task)
    })
      .then((response) => response.json())
  },

  deleteTask(cardId, task) {
    return fetch(`${API_URL}/cards/${cardId}/tasks/${task.id}`, {
      method: 'delete',
      headers: API_HEADERS
    })
  }
}

```

```

    })
  },
  toggleTask(cardId, task) {
    return fetch(`${API_URL}/cards/${cardId}/tasks/${task.id}`, {
      method: 'put',
      headers: API_HEADERS,
      body: JSON.stringify({done:!task.done})
    })
  }
}
};

export default KanbanAPI;

```

### 3. CardStore

该流程的最后一步更新 CardStore(见代码清单 6-53)。注意,除响应所有的 Action 来操作自身 state 之外,还创建了两个助手方法: getCard 和 getCardIndex。

#### 代码清单 6-53: 更新后的 CardStore

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {ReduceStore} from 'flux/utils';
import update from 'react-addons-update';
import 'babel-polyfill';

class CardStore extends ReduceStore {
  getInitialState() {
    return [];
  }

  getCard(id) {
    return this._state.find((card) => card.id == id);
  }

  getCardIndex(id) {
    return this._state.findIndex((card) => card.id == id);
  }

  reduce(state, action) {
    let cardIndex, taskIndex;

    switch (action.type) {
      case constants.FETCH_CARDS_SUCCESS:
        return action.payload.response;
    }

    /*
     * Card Creation
     */

```

```

    */
    case constants.CREATE_CARD:
      return update(this.getState(), {$push: [action.payload.card] })

    case constants.CREATE_CARD_SUCCESS:
      cardIndex = this.getCardIndex(action.payload.card.id);
      return update(this.getState(), {
        [cardIndex]: {
          id: {$set: action.payload.response.id}
        }
      });

    case constants.CREATE_CARD_ERROR:
      cardIndex = this.getCardIndex(action.payload.card.id);
      return update(this.getState(), {$splice: [[cardIndex, 1]]});
  }
  /*
  * Card Update
  */
  case constants.UPDATE_CARD:
    cardIndex = this.getCardIndex(action.payload.card.id);
    return update(this.getState(), {
      [cardIndex]: {
        $set: action.payload.draftCard
      }
    });

  case constants.UPDATE_CARD_ERROR:
    cardIndex = this.getCardIndex(action.payload.card.id);
    return update(this.getState(), {
      [cardIndex]: {
        $set: action.payload.card
      }
    });

  /*
  * Card Drag'n Drop
  */
  case constants.UPDATE_CARD_POSITION:
    if (action.payload.cardId !== action.payload.afterId) {
      cardIndex = this.getCardIndex(action.payload.cardId);
      let card = this.getState()[cardIndex];
      let afterIndex = this.getCardIndex(action.payload.afterId);
      return update(this.getState(), {
        $splice: [
          [cardIndex, 1],
          [afterIndex, 0, card]
        ]
      });
    }

```

```

    }

    case constants.UPDATE_CARD_STATUS:
      cardIndex = this.getCardIndex(action.payload.cardId);
      return update(this.getState(), {
        [cardIndex]: {
          status: { $set: action.payload.listId }
        }
      });

    case constants.PERSIST_CARD_DRAG_ERROR:
      cardIndex = this.getCardIndex(action.payload.cardProps.id);
      return update(this.getState(), {
        [cardIndex]: {
          status: { $set: action.payload.cardProps.status }
        }
      });
  });

  /*
  * Task Creation
  */
  case constants.CREATE_TASK:
    cardIndex = this.getCardIndex(action.payload.cardId);
    return update(this.getState(), {
      [cardIndex]: {
        tasks: { $push: [action.payload.task] }
      }
    });

  case constants.CREATE_TASK_SUCCESS:
    cardIndex = this.getCardIndex(action.payload.cardId);
    taskIndex = this.getState()[cardIndex].tasks.findIndex((task) => (
      task.id === action.payload.task.id
    ));
    return update(this.getState(), {
      [cardIndex]: {
        tasks: {
          [taskIndex]: {
            id: { $set: action.payload.response.id }
          }
        }
      }
    });

  case constants.CREATE_TASK_ERROR:
    let cardIndex = this.getCardIndex(action.payload.cardId);
    let taskIndex = this.getState()[cardIndex].tasks.findIndex(
      (task) => (
        task.id === action.payload.task.id
      )
    );

```



```

    ));
    return update(this.getState(), {
      [cardIndex]: {
        tasks: {
          $splice: [[taskIndex, 1]]
        }
      }
    });
  });
}

/*
 * Task Deletion
 */
case constants.DELETE_TASK:
  cardIndex = this.getCardIndex(action.payload.cardId);
  return update(this.getState(), {
    [cardIndex]: {
      tasks: { $splice: [[action.payload.taskIndex, 1]] }
    }
  });

case constants.DELETE_TASK_ERROR:
  cardIndex = this.getCardIndex(action.payload.cardId);
  return update(this.getState(), {
    [cardIndex]: {
      tasks: { $splice: [[action.payload.taskIndex, 0,
        action.payload.task]] }
    }
  });

/*
 * Task Toggling
 */
case constants.TOGGLE_TASK:
  cardIndex = this.getCardIndex(action.payload.cardId);
  return update(this.getState(), {
    [cardIndex]: {
      tasks: {
        [action.payload.taskIndex]: { done: { $apply: (done) => !done } }
      }
    }
  });

case constants.TOGGLE_TASK_ERROR:
  cardIndex = this.getCardIndex(action.payload.cardId);
  return update(this.getState(), {
    [cardIndex]: {
      tasks: {
        [action.payload.taskIndex]: { done: { $apply: (done) => !done } }
      }
    }
  });

```

```

    }
  });

  default:
    return state;
  }
}

export default new CardStore(AppDispatcher);

```

## 6.7.6 组件

现在回头分析组件, 让我们从 KanbanBoardContainer 组件中移除所有数据操作方法。通过观察可以发现这些方法被分组到两个对象中: taskActions 和 cardActions, 并以 props 形式传递给 KanbanBoard、List、Card 和 CheckList 组件。你必须从 KanbanBoardContainer 组件中移除这些方法、更改以上提到的所有组件的原型和 render 方法。你将不再以 props 形式向 NewCard 和 EditCard 组件传递卡片, 所以这两个组件也需要编辑。

### 1. KanbanBoardContainer

下面逐个处理这些组件。代码清单 6-54 展示了更新后的 KanbanBoardContainer 代码, 不包含构造函数和数据操作方法, 但包含了更新后的 render 方法(不再以 props 形式传递方法)。

代码清单 6-54: 更新后的 KanbanBoardContainer 组件

```

import React, { Component } from 'react';
import { Container } from 'flux/utils';
import KanbanBoard from './KanbanBoard';
import CardActionCreators from '../actions/CardActionCreators';
import CardStore from '../stores/CardStore';

class KanbanBoardContainer extends Component {
  componentDidMount() {
    CardActionCreators.fetchCards();
  }

  render() {
    let kanbanBoard = this.props.children && React.cloneElement(
      this.props.children, {
        cards: this.state.cards,
      });

    return kanbanBoard;
  }
}

KanbanBoardContainer.getStores = () => ([CardStore]);

```

```
KanbanBoardContainer.calculateState= (prevState) => ({
  cards: CardStore.getState()
});

export default Container.create(KanbanBoardContainer);
```

## 2. KanbanBoard

层级结构中的下一个是 KanbanBoard 组件。你不再向对象列表传递 taskCallbacks 和 cardCallbacks，也不再需要克隆子元素的属性(它们由 React Router 提供)。之前你克隆组件向其注入 props，但在 Flux 架构中，不再有此必要。代码清单 6-55 展示了更新后的 KanbanBoard 组件。

代码清单 6-55: 更新后的 KanbanBoard 组件

```
import React, { Component, PropTypes } from 'react';
import { DragDropContext } from 'react-dnd';
import HTML5Backend from 'react-dnd-html5-backend';
import { Link } from 'react-router';
import List from './List';

class KanbanBoard extends Component {
  render() {
    return (
      <div className="app">
        <Link to="/new" className="float-button"></Link>

        <List id='todo' title="To Do" cards={
          this.props.cards.filter((card) => card.status === "todo")
        } />

        <List id='in-progress' title="In Progress" cards={
          this.props.cards.filter((card) => card.status === "in-progress")
        } />

        <List id='done' title="Done" cards={
          this.props.cards.filter((card) => card.status === "done")
        } />

        {this.props.children}
      </div>
    )
  }
}

KanbanBoard.propTypes = {
  cards: PropTypes.arrayOf(PropTypes.object)
}

export default DragDropContext(HTML5Backend)(KanbanBoard);
```

### 3. 列表

层级结构中的下一个组件是List组件。你一样会移除taskCallbacks和cardCallbacks，但在这个文件中，还会更新listTargetSpec对象中的hover方法；它用来调用cardCallbacks.updateStatus，但现在它应该调用updateCardStatus Action创建器。代码清单6-56展示了更新后的源代码。

代码清单 6-56: 更新后的 List 组件

```
import React, { Component, PropTypes } from 'react';
import { DropTarget } from 'react-dnd';
import Card from './Card';
import constants from '../constants';
import CardActionCreators from '../actions/CardActionCreators';

const listTargetSpec = {
  hover(props, monitor) {
    const dragged = monitor.getItem();
    CardActionCreators.updateCardStatus(dragged.id, props.id);
  }
};

function collect(connect, monitor) {
  return {
    connectDropTarget: connect.dropTarget()
  };
}

class List extends Component {
  render() {
    const { connectDropTarget } = this.props;

    let cards = this.props.cards.map((card) => {
      return <Card key={card.id} {...card} />
    });

    return connectDropTarget(
      <div className="list">
        <h1>{this.props.title}</h1>
        {cards}
      </div>
    );
  }
}

List.propTypes = {
  id: PropTypes.string.isRequired,
  title: PropTypes.string.isRequired,
  cards: PropTypes.arrayOf(React.PropTypes.object),
  connectDropTarget: PropTypes.func.isRequired
}
```

```

}

export default DropTarget(constants.CARD, listTargetSpec, collect)(List);

```

#### 4. 卡片

是时候更新 Card 组件了。还是像之前所做的那样：移除 taskCallbacks 和 cardCallbacks 的所有引用，并将对这些 props 的调用更改为对 Action 创建器的调用。代码清单 6-57 展示了更新后的 Card 组件。

代码清单 6-57：更新后的 Card 组件

```

import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from '../constants';
import CheckList from './CheckList';
import { Link } from 'react-router';
import CardActionCreators from '../actions/CardActionCreators';

let titlePropType = (props, propName, componentName) => {...}

const cardDragSpec = {
  beginDrag(props) {
    return {
      id: props.id,
      status: props.status
    };
  },
  endDrag(props) {
    CardActionCreators.persistCardDrag(props);
  }
};

const cardDropSpec = {
  hover(props, monitor) {
    const draggedId = monitor.getItem().id;
    if(props.id !== draggedId){
      CardActionCreators.updateCardPosition(draggedId, props.id);
    }
  }
};

let collectDrag = (connect, monitor) => {...}
let collectDrop = (connect, monitor) => {...}

class Card extends Component {...}
Card.propTypes = {

```

```

    id: PropTypes.number,
    title: titlePropType,
    description: PropTypes.string,
    color: PropTypes.string,
    tasks: PropTypes.array,
    status: PropTypes.string,
    connectDragSource: PropTypes.func.isRequired,
    connectDropTarget: PropTypes.func.isRequired
  }

  const dragHighOrderCard = DragSource(constants.CARD, cardDragSpec,
    collectDrag)(Card);
  const dragDropHighOrderCard = DropTarget(constants.CARD, cardDropSpec,
    collectDrop)
    (dragHighOrderCard);
  export default dragDropHighOrderCard

```

## 5. CheckList

在 CheckList 组件中，去掉所有的 TaskCallback 调用，并增加 TaskActionCreator 调用。代码清单 6-58 展示了更新后的代码。

### 代码清单 6-58：更新后的 CheckList 组件

```

import React, { Component, PropTypes } from 'react';
import TaskActionCreators from '../actions/TaskActionCreators';

class CheckList extends Component {
  checkInputKeyPress(evt) {
    if (evt.key === 'Enter') {
      let newTask = {id: Date.now(), name: evt.target.value, done: false};
      TaskActionCreators.addTask(this.props.cardId, newTask);
      evt.target.value = '';
    }
  }

  render() {
    let tasks = this.props.tasks.map((task, taskIndex) => (
      <li key={task.id} className="checklist__task">
        <input type="checkbox"
          checked={task.done}
          onChange={
            TaskActionCreators.toggleTask.bind(null, this.props.cardId,
              task, taskIndex)
          } />
        {task.name}{' ' }
        <a href="#"
          className="checklist__task--remove"
          onClick={

```



```

        TaskActionCreators.deleteTask.bind(null, this.props.cardId,
        task, taskIndex)
      } />
    </li>
  );
}

return (
  <div className="checklist">
    <ul>{tasks}</ul>
    <input type="text"
      className="checklist--add-task"
      placeholder="Type then hit Enter to add a task"
      onKeyDown={this.checkInputKeyPress.bind(this)} />
    </div>
  );
}

CheckList.propTypes = {
  cardId: PropTypes.number,
  tasks: PropTypes.arrayOf(PropTypes.object)
}

export default CheckList;

```

## 6. NewCard 和 EditCard

最后更新 NewCard 和 EditCard 这两个组件。在这两个组件中，都需要用 Action 创建器调用来替换以 props 的形式传入的回调。在 EditCard 中，你还会再进一步：该组件不再定义卡片数组 props，你会直接与 CardStore 通信来获取选中的卡片明细。代码清单 6-59 展示了更新后的 NewCard 组件，代码清单 6-60 展示了更新后的 EditCard 组件。

### 代码清单 6-59：更新后的 NewCard.js

```

import React, {Component, PropTypes} from 'react';
import CardForm from './CardForm';
import CardActionCreators from '../actions/CardActionCreators';

class NewCard extends Component {
  componentWillMount() {...}
  handleChange(field, value){...}

  handleSubmit(e) {
    e.preventDefault();
    CardActionCreators.addCard(this.state);
    this.props.history.pushState(null, '/');
  }

  handleClose(e) {
    this.props.history.pushState(null, '/');
  }
}

```

```

    render() {...}
  }
  NewCard.propTypes = {...};

  export default NewCard;

```

#### 代码清单 6-60: 更新后的 EditCard.js

```

import React, {Component, PropTypes} from 'react';
import CardForm from '../CardForm';
import CardStore from '../stores/CardStore';
import CardActionCreators from '../actions/CardActionCreators';
import 'babel-polyfill'

class EditCard extends Component {
  componentWillMount() {
    let card = CardStore.getCard(parseInt(this.props.params.card_id));
    this.setState(Object.assign({}, card));
  }

  handleChange(field, value) {...}

  handleSubmit(e) {
    e.preventDefault();
    CardActionCreators.updateCard(CardStore.getCard(parseInt(
      this.props.params.card_id)),
      this.state);
    this.props.history.pushState(null, '/');
  }

  handleClose(e) {...}

  render() {...}
}

EditCard.propTypes = {...};

export default EditCard;

```

### 6.7.7 删除所有组件 state

理想情况下,应该避免在使用 Flux 的同时还使用和操作组件的 state。所有组件状态(甚至是 UI 相关的状态)都应该被保存在 Store 中。这是编写 Flux 应用程序的最佳实践和可取目标,但让有状态的组件在有限的范围内使用小巧、UI 相关的数据并算不上什么本质的错误。

目前的看板应用即是如此。你几乎将所有内容都转到 Flux 架构,但有些组件依然拥

有私有状态。Card 组件保存了一个 showDetails 私有状态，EditCard 和 NewCard 组件也有私有状态(用来保存将要被操作的草稿卡片)。正如我们所言，这并没有错，但出于完全向 Flux 迁移的目的，还是让我们将所有私有状态都移到 Store，也让组件更精简一些。

## 1. 显示/隐藏卡片明细

让我们从 Card 组件的 showDetails 开始。你不会将数据持久化到服务器，但你会使用现有的 CardStore 来保存这个值。CardStore 将从 KanbanApi 加载而来的卡片数据设置到其 state 中。你需要为每张卡片增加一个 ShowProperties 键，但为简单起见，你不会在获取初始数据时执行这一操作。相反，你会为没有设置这一属性的卡片假定一个默认值，并仅在用户切换了卡片明细可见性时才设置这一属性。用普通语言表述为：如果卡片没有 showDetails 属性，就假设其明细将要显示。当用户首次关闭卡片明细时，你就会为对应的卡片创建这一属性，并将其值设置为 false。

### 卡片组件

从 Card 组件开始，你将：

- 摆脱构造函数(因为不需要设置初始 state)。
- 当用户尝试切换明细可见性时调用 Action 创建器。
- 将对 this.state.showDetails 的所有引用更改为 this.props.showDetails。
- 确保仅在属性存在时才显示明细(通过检查它是否被显式设置为 false)。

代码清单 6-61 展示了更新后的 Card 组件。

### 代码清单 6-61：更新后的 Card 组件没有私有状态

```
import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from '../constants';
import CheckList from './CheckList';
import { Link } from 'react-router';
import CardActionCreators from '../actions/CardActionCreators';

let titlePropType = (props, propName, componentName) => {...};
const cardDragSpec = {...};
const cardDropSpec = {...};
let collectDrag = (connect, monitor) => {...};
let collectDrop = (connect, monitor) => {...};

class Card extends Component {
  toggleDetails() {
    CardActionCreators.toggleCardDetails(this.props.id);
  }
}
```

```

render() {
  const { isDragging, connectDragSource, connectDropTarget } = this.props;

  let cardDetails;
  if (this.props.showDetails !== false) {
    cardDetails = (...);
  }

  let sideColor = {...};

  return connectDropTarget(connectDragSource(
    <div className="card" >
      <div style={sideColor}/>
      <div className="card__edit"><Link to={'/edit/'+this.props.id}>
        </Link></div>
      <div className={
        this.props.showDetails !== false? "card__title card__
        title--is-open" :
        "card__title"
      } onClick={this.toggleDetails.bind(this)}>
        {this.props.title}
      </div>
      <ReactCSSTransitionGroup transitionName="toggle"
        transitionEnterTimeout={250}
        transitionLeaveTimeout={250}>
        {cardDetails}
      </ ReactCSSTransitionGroup>
    </div>
  ));
}
}
Card.propTypes = {...};

const dragHighOrderCard = DragSource(CARD, cardDragSpec, collectDrag)(Card);
const dragDropHighOrderCard = DropTarget(CARD, cardDropSpec, collectDrop)
(dragHighOrderCard);
export default dragDropHighOrderCard

```

### 常量和 Action 创建器

接下来，将实现 toggleCardDetails Action 创建器。需要一个常量来标识这个 Action，所以在 constants.js 文件中添加一个新的 TOGGLE\_CARD\_DETAILS，如代码清单 6-62 所示。

代码清单 6-62: constants.js 文件的部分内容，包含 TOGGLE\_CARD\_DETAILS

```

export default{
  CARD:'card',

  FETCH_CARDS:'fetch cards',

```

```

    FETCH_CARDS_SUCCESS: 'fetch cards success',
    FETCH_CARDS_ERROR: 'fetch cards error',
    ...

    TOGGLE_CARD_DETAILS: 'toggle card details',
    ...

    TOGGLE_TASK: 'toggle task',
    TOGGLE_TASK_SUCCESS: 'toggle task success',
    TOGGLE_TASK_ERROR: 'toggle task error',
  };

```

按照顺序，让我们来编辑 CardActionCreator 文件，如代码清单 6-63 所示。

代码清单 6-63：更新后的 CardActionCreator 包含 toggleCardDetails 方法

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';
import {throttle} from '../utils';
import CardStore from '../stores/CardStore';

let CardActionCreators = {

  fetchCards() {...},

  toggleCardDetails(cardId) {
    AppDispatcher.dispatch({
      type: constants.TOGGLE_CARD_DETAILS,
      payload: {cardId}
    });
  },

  addCard(card) {...},

  updateCard(card, draftCard) {...},

  updateCardStatus: throttle((cardId, currListId, nextListId) => {...}),
  updateCardPosition: throttle((cardId, afterId) => {...}, 500),

  persistCardDrag(cardProps) {...}
};

export default CardActionCreators;

```

## CardStore

最后更新 CardStore。注意你会显式地检查 showDetails 的值是否等于 false(检查失败, 表示该属性尚未设置)。代码清单 6-64 展示了更新后的文件。

### 代码清单 6-64: 更新后的 CardStore

```
import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {ReduceStore} from 'flux/utils';
import update from 'react-addons-update';
import 'babel-polyfill'

class CardStore extends ReduceStore {
  getInitialState() {...}
  getCard(id) {...}
  getCardIndex(id) {...}

  reduce(state, action){
    let cardIndex, taskIndex;

    switch (action.type) {
      case constants.FETCH_CARDS_SUCCESS:
        ...

      /*
       * Card Creation
       */
      case constants.CREATE_CARD:
        ...
      case constants.CREATE_CARD_SUCCESS:
        ...
      case constants.CREATE_CARD_ERROR:
        ...

      /*
       * Card Status Toggle
       */
      case constants.TOGGLE_CARD_DETAILS:
        cardIndex = this.getCardIndex(action.payload.cardId);
        return update(this.getState(), {
          [cardIndex]: {
            showDetails: {$apply: (currentValue) =>
              (currentValue !== false)?false : true)
          }
        });

      /*
       * Card Update
       */
    }
```



```

    case constants.UPDATE_CARD:
      ...
    case constants.UPDATE_CARD_ERROR:
      ...

    /*
     * Card Drag'n Drop
     */
    case constants.UPDATE_CARD_POSITION:
      ...
    case constants.UPDATE_CARD_STATUS:
      ...
    case constants.PERSIST_CARD_DRAG_ERROR:
      ...

    /*
     * Task Creation
     */
    case constants.CREATE_TASK:
      ...
    case constants.CREATE_TASK_SUCCESS:
      ...
    case constants.CREATE_TASK_ERROR:
      ...

    /*
     * Task Deletion
     */
    case constants.DELETE_TASK:
      ...
    case constants.DELETE_TASK_ERROR:
      ...

    /*
     * Task Toggling
     */
    case constants.TOGGLE_TASK:
      ...
    case constants.TOGGLE_TASK_ERROR:
      ...

    default:
      return state;
  }
}

export default new CardStore(AppDispatcher);

```

## 2. 编辑和新建卡片的组件

仅剩的两个拥有私有状态的组件是 EditCard 和 NewCard。就这两者的情况而言，其

私有状态要比单个属性更复杂。它保存了完整的卡片结构。因此，你会创建一个全新的 Store: DraftStore，并用它来保存将要被编辑的卡片信息。

### DraftStore

DraftStore响应两个Action: CREATE\_DRAFT和UPDATE\_DRAFT。当CREATE\_DRAFT Action被分发后，DraftStore就会将其内部state更新为空卡片对象(针对创建新卡片的情况)或者现有卡片对象的副本(针对编辑卡片的情况)。会向一个受控的表单提供草稿卡片，每当其发生变化时，都会分发一个UPDATE\_DRAFT Action。

在开始迁移其余的实现时，先来看看 DraftStore 的源代码(见代码清单 6-65)。

### 代码清单 6-65: 新的 DraftStore 源代码

```
import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import {ReduceStore} from 'flux/utils';
import update from 'react-addons-update';

let defaultDraft = () =>{
  return{
    id:Date.now(),
    title:'',
    description:'',
    status:'todo',
    color:'#c9c9c9',
    tasks:[]
  }
};

class DraftStore extends ReduceStore {
  getInitialState() {
    return{};
  }

  reduce(state, action){
    switch (action.type) {
      case constants.CREATE_DRAFT:
        if(action.payload.card){
          return update(this.getState(),{
            $set:action.payload.card
          });
        }else{
          return defaultDraft();
        }
      case constants.UPDATE_DRAFT:
        return update(this.getState(),{
          [action.payload.field]:{
            $set:action.payload.value
          }
        });
    }
  }
}
```

```
    }  
  });  
  
  default:  
    return state;  
}  
}  
  
export default new DraftStore(AppDispatcher);
```

这段代码里有一些内容值得注意。首先定义了一个名为 `defaultDraft` 的函数，它会返回一个默认的、干净的卡片对象，并拥有一个临时 ID。

此外，在 `reduce` 方法的 `switch` 语句中，在响应 `CREATE_DRAFT` Action 时，你会检查是否以 `payload` 形式传入了一个卡片对象。这对应了编辑现有卡片的情况；接着，卡片属性会被复制并设置为 `Store` 的 `state`。如果没有通过参数传入卡片(对应用户创建新卡片的情况)，就会调用 `defaultDraft` 方法来创建一个默认空卡片，并将其设置为 `Store` 的 `state`。

`UPDATE_DRAFT` Action 会传递两个 `payload`：用户所编辑的 `field` 及其新 `value`。在本例中，新 `value` 会被设置到 `Store` 的 `state` 中的草稿卡片的对应属性中。

常量和 Action 创建器

这里并没有什么特别值得注意的。仅添加了一些新常量，并声明了新的 Action 创建器，分别如代码清单 6-66 和代码清单 6-67 所示。

代码清单 6-66：更新后的 `constants.js` 文件的部分源代码，包含 `CREATE_DRAFT` 和 `UPDATE_DRAFT` 常量

```
export default({  
  CARD: 'card',  
  
  FETCH_CARDS: 'fetch cards',  
  FETCH_CARDS_SUCCESS: 'fetch cards success',  
  FETCH_CARDS_ERROR: 'fetch cards error',  
  
  ...  
  
  CREATE_DRAFT: 'create draft',  
  UPDATE_DRAFT: 'update draft',  
  
  ...  
  
  TOGGLE_TASK: 'toggle task',  
  TOGGLE_TASK_SUCCESS: 'toggle task success',  
  TOGGLE_TASK_ERROR: 'toggle task error',  
});
```

代码清单 6-67: 更新后的 CardActionCreators.js

```

import AppDispatcher from '../AppDispatcher';
import constants from '../constants';
import KanbanAPI from '../api/KanbanApi';
import {throttle} from '../utils';
import CardStore from '../stores/CardStore';

let CardActionCreators = {

  fetchCards() {...},
  toggleCardDetails(cardId) {...},
  addCard(card) {...},
  updateCard(card, draftCard) {...},
  updateCardStatus:throttle((cardId, currListId, nextListId) =>{...}),
  updateCardPosition:throttle((cardId , afterId) =>{...},500),
  persistCardDrag(cardProps) {...},

  createDraft(card) {
    AppDispatcher.dispatch({
      type:constants.CREATE_DRAFT,
      payload:{card}
    });
  },

  updateDraft(field, value) {
    AppDispatcher.dispatch({
      type:constants.UPDATE_DRAFT,
      payload:{field, value}
    });
  }
};

export default CardActionCreators;

```

### EditCard 和 NewCard 组件

为完成从组件中移除私有状态这一目标,让我们来更新 EditCard 和 NewCard 这两个文件。你会移除构造函数方法,并将所有私有状态操作替换为对 Action 创建器的调用。此外,这两个组件还都会使用 Flux 库的 Container 高阶函数来侦听 DraftStore 的变化,并映射到其 state。代码清单 6-68 和代码清单 6-69 展示了更新后的代码。

代码清单 6-68: 更新后的 EditCard 组件

```

import React,{Component} from 'react';
import CardForm from './CardForm';
import CardStore from '../stores/CardStore';
import DraftStore from '../stores/DraftStore';
import {Container} from 'flux/utils';

```

```

import CardActionCreators from '../actions/CardActionCreators';

class EditCard extends Component {
  handleChange(field, value) {
    CardActionCreators.updateDraft(field, value);
  }

  handleSubmit(e) {
    e.preventDefault();
    CardActionCreators.updateCard(
      CardStore.getCard(this.props.params.card_id), this.state.draft
    );
    this.props.history.pushState(null, '/');
  }

  handleClose(e) {
    this.props.history.pushState(null, '/');
  }

  componentDidMount() {
    setTimeout(() => {
      CardActionCreators.createDraft(CardStore.getCard(
        this.props.params.card_id)
      ), 0);
    }
  )

  render() {
    return (
      <CardForm draftCard={this.state.draft}
        buttonLabel="Edit Card"
        handleChange={this.handleChange.bind(this)}
        handleSubmit={this.handleSubmit.bind(this)}
        handleClose={this.handleClose.bind(this)} />
    )
  }
}

EditCard.getStores = () => ([DraftStore]);
EditCard.calculateState = (prevState) => ({
  draft: DraftStore.getState()
});

```

```
export default Container.create(EditCard);
```

代码清单 6-69: 更新后的 NewCard 组件

```

import React, {Component} from 'react';
import CardForm from './CardForm';
import DraftStore from '../stores/DraftStore';

```

```

import {Container} from 'flux/utils';
import CardActionCreators from '../actions/CardActionCreators';

class NewCard extends Component{
  handleChange(field, value){
    CardActionCreators.updateDraft(field, value);
  }

  handleSubmit(e){
    e.preventDefault();
    CardActionCreators.addCard(this.state.draft);
    this.props.history.pushState(null, '/');
  }

  handleClose(e){
    this.props.history.pushState(null, '/');
  }

  componentDidMount(){
    setTimeout(()=>CardActionCreators.createDraft(), 0)
  }

  render(){
    return (
      <CardForm draftCard={this.state.draft}
        buttonLabel="Create Card"
        handleChange={this.handleChange.bind(this)}
        handleSubmit={this.handleSubmit.bind(this)}
        handleClose={this.handleClose.bind(this)} />
    )
  }
}

NewCard.getStores = () => ([DraftStore]);
NewCard.calculateState = (prevState) => ({
  draft: DraftStore.getState()
});

export default Container.create(NewCard);

```

这确实是一次大型的项目重构，但最终得到一个更清晰、更加结构化且可被预测的代码库。和往常一样，完整的源代码可在 Apress 网站([www.apress.com](http://www.apress.com))以及本书的 GitHub 页面([pro-react.github.io](http://pro-react.github.io))上找到。

## 6.8 本章小结

在本章中，你了解了什么是 Flux，以及它解决了哪些问题。你看到了如何在 React 应用程序中集成 Flux，以及如何构建包含异步 API 通信的复杂应用程序。



性能调优

从底层设计开始，React 就已经充分考虑到性能问题。在更新 UI 时，它使用了一些很聪明的技巧来尽可能减少耗时的 DOM 操作。此外，它也提供了一些工具和方法，能在我们需要时对性能做进一步优化。

在本章中，你将学习到 React 中的“子级校正”(Reconciliation)过程是如何工作的；如何使用 React Perf 来判别性能瓶颈；如何在组件中使用 `shouldComponentUpdate` 生命周期中的方法来尽快完成界面重绘过程，从而对性能进行提升。

7.1 子级校正过程的工作原理

当你改变 React 组件状态时，它会触发组件的重绘过程。React 会构建一个新的虚拟 DOM 来呈现应用 UI 的状态，然后检测和当前的虚拟 DOM 之间的差异，从而计算出哪些 DOM 元素需要进行更新、添加或删除。这个过程被称为“子级校正(reconciliation)”。

7.1.1 批处理

在 React 中，在任何时刻调用组件的 `setState` 方法，React 不会立即对其更新，而是只会将其标记为“脏”状态(图 7-1 中展示了这一过程)。这也就意味着，组件状态的变更不会立刻生效，React 使用了事件轮询对变更内容进行批量绘制。

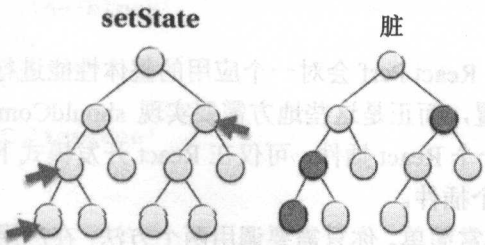


图 7-1 当你调用组件的 `setState` 方法时，React 仅将其标记为脏状态

事件轮询是一个 JavaScript 机制，它会持续运行，不断地进行数据分发，检测所有需要被触发的事件处理程序和生命周期方法。通过对子级校正过程进行批处理，在每个事件轮询中，DOM 节点只会进行一次更新，这正是实现一个高性能应用的关键所在。

7.1.2 子树渲染

当事件轮询结束后，React 会将“脏”组件及其子节点进行重绘。所有后代节点的 render 方法都会被调用，哪怕它们并没有发生变化，如图 7-2 所示。

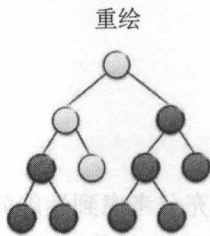


图 7-2 在一个事件轮询的最后，脏组件 DOM 树会在虚拟 DOM 中进行重绘

这个过程看上去效率很低，不过实际上它非常快，因为 React 并没有直接操作实际的 DOM。整个过程都发生在内存中的虚拟 DOM 上，而在现代浏览器中，JavaScript 处理这类操作的速度已经超级快了。

然而，React 还提供了一种方法对这个过程进行优化，阻止子树的重绘：通过一个名为 `shouldComponentUpdate` 的生命周期方法。在子组件重绘之前，React 都会调用它的 `shouldComponentUpdate` 方法。默认情况下，该方法始终返回 `true`，不过如果你自己实现了该方法并返回 `false`，React 就会跳过该组件及其子组件的重绘过程。

需要注意，React 默认情况下已经足够快了，只有在个别情况下需要使用这个方法。在不必要的情况下使用 `shouldComponentUpdate` 方法相当于过早的优化，这是一个糟糕的实践，不仅浪费了时间，还增加了代码的复杂度从而导致更多可能出现的 bug，尤其是它很难进行调试。与盲目在应用组件中实现 `shouldComponentUpdate` 方法相比，最好的方式则是对你的应用进行分析，检查它是否需要进行性能的调整，以及在哪里进行调整。

7.2 React Perf

作为一个分析工具，React Perf 会对一个应用的整体性能进行概要性分析，帮助我们发现需要进行优化的位置，而正是这些地方需要实现 `shouldComponentUpdate` 这一生命周期方法。Perf 对象是一个 React 插件，可仅在 React 开发模式下使用。你不应该在构建生产环境应用时包含这个插件。

React Perf 的 API 非常简单。你只需要调用两个方法：在应用中开始测量的地方调用 `Perf.start()`，在完成测量的地方调用 `Perf.stop()`。React Perf 模块还提供了 3 个方法来显示相关数据，这些数据会在测量完成后用格式化表格的形式显示在浏览器的控制台中，详见表 7-1。

表 7-1 React Perf 方法

校验器	说明
Perf.start()和 Perf.stop()	开始/结束测量。在这两者之间的 React 操作会记录下来以供后续方法分析
Perf.printInclusive()	打印消耗的全部时间
Perf.printExclusive()	“排他”的时间中不包括挂载组件所消耗的时间，包括处理属性、调用 componentWillMount 和 componentDidMount 等
Perf.printWasted()	“浪费”时间表示在组件中实际上没有做任何渲染的消耗。换句话说，渲染的内容是保持不变的，所以实际上并没有对 DOM 进行操作

提示：

虽然 React Perf 插件提供了一些很有价值的洞察分析，但它仍然不可能检测到你的应用中所有可以优化的点。请配合使用浏览器的开发工具，自己对应用进行检测和调试。

7.2.1 性能测试应用

为对 React Perf 进行实验，并在接下来实现 shouldComponentUpdate 方法，我们来创建一个简单的 React 应用：时钟。这个应用包含 3 个组件：主 App 组件、Clock 组件和 Digit 组件。

下面自下向上地进行开发。Digit 组件会接收一个数字属性，判断它是否小于 10(如果小于 10 的话在前面补 0)，然后渲染这个数字。代码清单 7-1 展示了完整代码。

代码清单 7-1: Digit.js 源代码

```
import React, { Component, PropTypes } from 'react';

class Digit extends Component {
  render() {
    let digitStyle={
      display:'inline-block',
      fontSize: 20,
      padding: 10,
      margin: 5,
      background: '#eeeeee'
    };

    let displayValue;
    if(this.props.value < 10){
      displayValue = '0' + this.props.value;
    } else {
      displayValue = this.props.value;
    }
  }
}
```

```

    return (
      <div style={digitStyle}>{displayValue}</div>
    );
  }
}

Digit.propTypes = {
  value: PropTypes.number.isRequired
}

export default Digit;

```

接下来创建 Clock 组件。它会接收 3 个属性(hours、minutes 和 seconds), 然后依次渲染 3 个 Digit 组件。代码清单 7-2 中展示了 Clock.js 代码。

代码清单 7-2: Clock.js 代码

```

import React, { Component, PropTypes } from 'react'
import Digit from './Digit';

class Clock extends Component {
  render() {
    return (
      <div>
        <Digit value={this.props.hours} />{' : '}
        <Digit value={this.props.minutes} />{' : '}
        <Digit value={this.props.seconds} />
      </div>
    );
  }
}

Clock.propTypes = {
  hours: PropTypes.number.isRequired,
  minutes: PropTypes.number.isRequired,
  seconds: PropTypes.number.isRequired
}

export default Clock;

```

最后实现 App 组件。这是一个带有状态的组件, 包含一个 getTime 方法, 它会返回一个对象, 其中包括一些独立的属性 hours、minutes、seconds 和 milliseconds。该方法会在构造函数(用来初始化组件的状态)以及 componentDidMount 生命周期(设置应用状态并重复更新其数值)中进行调用。在 render 函数中, 显示一个 Clock 组件。代码清单 7-3 中展示了完整的代码。

代码清单 7-3: App 组件

```

import React, { Component } from 'react';

```

```

import { render } from 'react-dom';
import Clock from './Clock';

class App extends Component {
  constructor() {
    super(...arguments);
    this.state = this.getTime();
  }

  componentDidMount() {
    setInterval(() => {
      this.setState(this.getTime());
    }, 500);
  }

  getTime() {
    let now = new Date();
    return {
      hours: now.getHours(),
      minutes: now.getMinutes(),
      seconds: now.getSeconds()
    };
  }

  render() {
    return (
      <div>
        <Clock hours={this.state.hours}
          minutes={this.state.minutes}
          seconds={this.state.seconds} />
      </div>
    );
  }
}

render(<App />, document.getElementById("root"));

```

测试一下这个应用，你会看到类似图 7-3 中显示的内容。

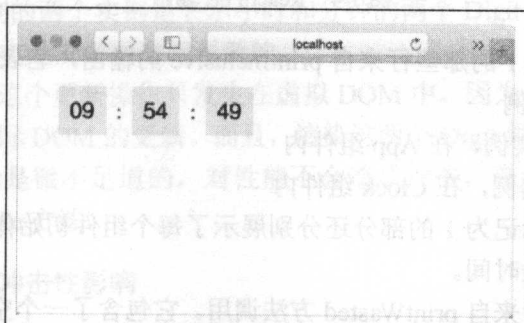


图 7-3 性能测试应用



7.2.2 安装并使用 React Perf

现在你已经完成了示例应用，下面安装ReactPerf，然后分析一下有哪些可以进行性能优化的地方。React Perf是一个插件，所以在继续之前请使用npm install --save react-addons-perf进行安装。

接下来，在 App 组件中导入 Perf 模块。你将在 app 即将渲染之前开始进行性能测量，然后在渲染完毕之后马上结束测量。接着调用 printInclusive(展示所有渲染的组件实例的列表及其消耗的时间)和 printWasted(列出那些在渲染中没有发生任何变化的组件实例)。更新后的代码如代码清单 7-4 所示。

代码清单 7-4：使用 React Perf 对性能测试应用进行分析

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import Perf from 'react-addons-perf';
import Clock from './Clock';

class App extends Component {...}

Perf.start();
render(<App />, document.getElementById("root"));
Perf.stop();
Perf.printInclusive();
Perf.printWasted();
```

现在，在浏览器中测试一下这个应用，在图 7-4 中可以看到控制台内的输出。

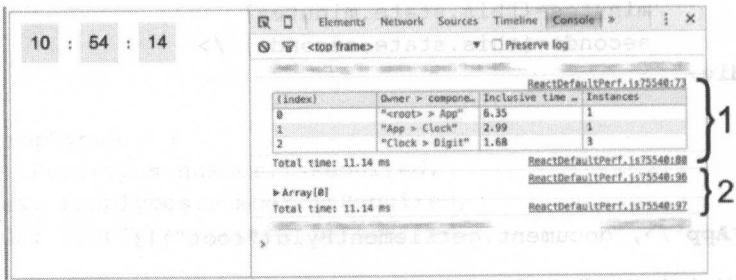


图 7-4 React Perf 输出

在图 7-4 中，标记 1 的那些行来自 printInclusive 的输出，它表示你的应用中：

- 有一个 App 实例
- 有一个 Clock 实例，在 App 组件内
- 有三个 Digit 实例，在 Clock 组件内

在控制台输出中标记为 1 的部分还分别展示了每个组件初始化和渲染的时间，以及应用总的初始化及渲染时间。

标记为 2 的那些行来自 printWasted 方法调用。它包含了一个空数组，因为该方法没有发现任何“浪费”的时间。



不过你所使用的测量方法有一个问题：在初次渲染之后就立即停止了分析，所以你的分析没有针对任何状态变化的过程。你只是对应用的初始状态进行了一个快照分析。为修正这个问题，将使用一个一秒多一点的定时器，在输出结果之前对应用进行测量。代码清单 7-5 展示了更新后的代码，图 7-5 展示了浏览器控制台中新的输出内容。

代码清单 7-5：在展示结果之前，用刚刚超过一秒的定时器进行分析

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import Clock from './Clock';
import Perf from 'react-addons-perf';

class App extends Component {...}
Perf.start()
render(<App />, document.getElementById("root"));
setTimeout(()=>{
  Perf.stop();
  Perf.printWasted();
},1500)
```

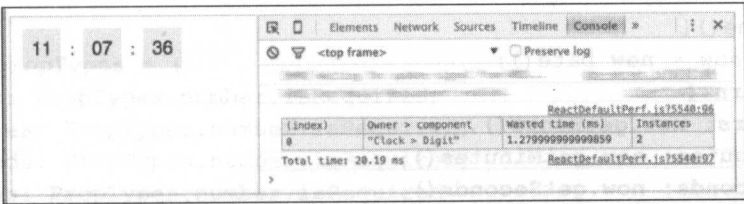


图 7-5 在一秒多之后的测量，printWasted 方法的输出

此外，注意上面的代码中只保留了 printWasted 的输出。

现在你可以注意到，React Perf 检测到了两个 Digit 组件不必要的渲染。这里发生了什么？为理解这个过程，下面重现一下事情的经过：

- 在时钟指向 11:07:35 时开始了测试。
- 在分析过程中，状态发生了变化，从而触发了 Clock 组件的重绘，显示当前时间是 11:07:36。
- Clock 组件因此渲染了全部三个 Digit 组件，即使其中有两个没有发生变化。

React Perf 检测到的两个实例是表示小时和分钟的两个 Digit 组件，因为它们的值没有发生变化，所以其实是没必要进行更新的。

不过也要注意，这个更新操作只发生在虚拟 DOM 中。因为 React 的差异处理过程，这些变化不会造成实际 DOM 的更新。而且，渲染这两个 Digit 组件所“浪费”的时间还不到两毫秒，可认为是微不足道的，对性能不会造成冲击。在这里绝对没有必要实现 shouldComponentUpdate 方法。

强行对性能造成冲击性影响

下面在应用中做一点改变，来引发一个性能问题：在 Clock 组件中增加一个十分之

一秒的域，每隔十分之一秒就对 App 组件进行更新；此外，为对性能造成冲击性影响，我们还要在屏幕上渲染 200 个时钟。代码清单 7-6 中展示了更新后的 App 组件，代码清单 7-7 中展示了更新后的 Clock 组件。Digit 组件不需要做修改。

代码清单 7-6：更新后的 App 组件，每隔十分之一秒运行一次，渲染 200 个时钟

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import Clock from './Clock';
import Perf from 'react-addons-perf';

class App extends Component {
  constructor() {...}

  componentDidMount() {
    setInterval(() => {
      this.setState(this.getTime());
    }, 10);
  }

  getTime() {
    let now = new Date();
    return {
      hours: now.getHours(),
      minutes: now.getMinutes(),
      seconds: now.getSeconds(),
      tenths: parseInt(now.getMilliseconds()/10),
    };
  }

  render() {
    let clocks=[];
    for (var i = 0; i < 200; i++) {
      clocks.push(<Clock hours={this.state.hours}
        minutes={this.state.minutes} seconds={this.state.seconds}
        tenths={this.state.tenths} />)
    }

    return (
      <div>
        {clocks}
      </div>
    );
  }
}

Perf.start()
render(<App />, document.getElementById("root"));
setTimeout(()=>
```

```
Perf.stop();
Perf.printWasted();
},2000)
```

代码清单 7-7：更新后的 Clock 组件

```
import React, { Component, PropTypes } from 'react'
import Digit from './Digit';

class Clock extends Component {
  render() {
    return (
      <div>
        <Digit value={this.props.hours} />{' : '}
        <Digit value={this.props.minutes} />{' : '}
        <Digit value={this.props.seconds} />{' . '}
        <Digit value={this.props.tenths} />
      </div>
    );
  }
}

Clock.propTypes = {
  hours: PropTypes.number.isRequired,
  minutes: PropTypes.number.isRequired,
  seconds: PropTypes.number.isRequired,
  tenths: PropTypes.number.isRequired
}

export default Clock;
```

现在，在浏览器中运行一下。你会注意到应用的性能变得有些迟缓。这是我们想要达到的目的，不过如果你在一台速度很快的机器上运行，可能看起来不会有什么不同，可以试着把循环次数增加一些。React Perf 确认了这一点：现在有很多时间浪费在那些没有发生变化的 Digit 组件的运算过程中。如图 7-6 所示，现在我们有了一个不得不关注的性能问题，浪费的时间超过了 0.6 秒。作为参考，FPS 帧率也显示出来，当前应用的运行速度是 15fps。

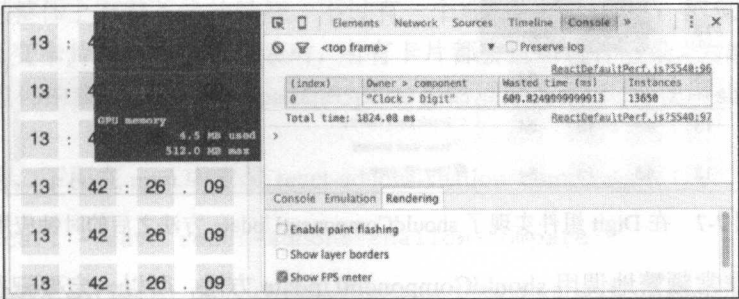


图 7-6 React Perf 显示在没有变化的 Digit 组件计算中，消耗了 609 毫秒

### 7.3 shouldComponentUpdate

React 提供了 `shouldComponentUpdate` 这一生命周期中的方法，它会在渲染开始之前调用，从而给我们提供了跳过整个组件树的渲染计算过程的可能。该方法接收 `nextProps` 和 `nextState` 作为参数，你需要返回 `true` 或 `false` 来告诉 React 该组件是否需要重新绘制。默认情况下它会返回 `true`，不过如果你返回 `false`，那么 React 将认为这个组件没发生变化，因此不会进行差异检测和重绘。

在时钟应用程序中，在 `Digit` 组件中实现 `shouldComponentUpdate` 方法时，你需要做的只是比较新旧两个属性值，如代码清单 7-8 所示。

代码清单 7-8：在 `Digit` 组件中实现 `shouldComponentUpdate`

```
import React, { Component, PropTypes } from 'react'

class Digit extends Component {

  shouldComponentUpdate(nextProps, nextState) {
    // Don't trigger a re-render unless the digit value has changed
    return nextProps.value !== this.props.value;
  }

  render() {...}
}

Digit.propTypes = {...}

export default Digit;
```

如图 7-7 所示，React Perf 在输出“wasted”渲染实例时，已经返回空数组了。性能的改善在浏览器中很容易就可以体会到(为证明这一点，我们再次显示了 FPS 值，现在速度已经是之前的两倍了)。

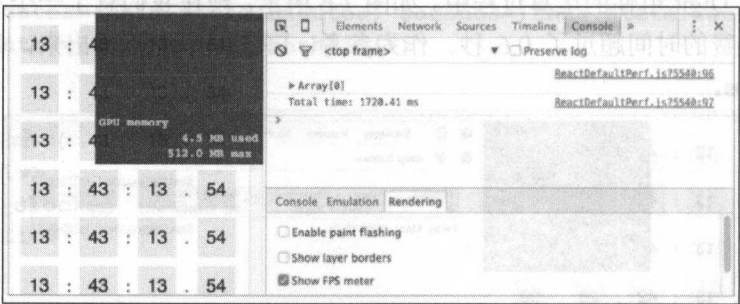


图 7-7 在 `Digit` 组件实现了 `shouldComponentUpdate` 方法之后的时钟应用

React 会非常频繁地调用 `shouldComponentUpdate` 方法，所以一定要记得在实现该方法时所有的检测、比较必须非常快，否则通过这一方法来改善应用性能就没什么意义了。

比较两个简单的值(就像前面的例子中做的)会非常快,所以这个方法能够行得通,但比较两个对象中层级很深的属性开销就很大了,这种方式就没什么意义了。

这就是使用不可变值的意义:它会让整个对象的跟踪和比较开销很小、速度极快而且高度可靠。

在第3章中,你曾经学过 React 中的不可变助手。它们会在 JavaScript 对象上实现不可变性,在改变时会返回带有修改后属性值的全新对象,而不是直接修改原有对象内部的值。这意味着在新旧对象之间进行浅度比较就足以判断它们是否有更新了,即使更新发生在很深的层级结构中。

#### 提示:

React 中的不变性助手提供了一个很优雅的机制,把 JavaScript 中默认的数据结构(它们并非不可变的)处理为不可变的,你可能还会想到需要一个库在 JavaScript 中提供“真正”不可变的集合。使用不可变的数据结构不仅会为 React 带来更好的性能,也会给你带来更好的数据一致性,并帮助你提高代码的质量。

有很多不同的库都可在 JavaScript 中实现不可变集合,包括 Facebook 自己的 Immutable-js。

Immutable-js 实现了一组高效的不可变数据结构,如 Lists、Maps、Sets 等。更多关于 Immutable-js 的信息请访问这个库的网站: <https://facebook.github.io/immutable-js/>。

## shallowCompare 插件

React 提供了一个名为 shallowCompare 的插件,可配合 shouldComponentUpdate 一起使用。它的浅度比较针对组件的 props 和 state 属性,并返回该组件是否发生了变化。

shallowCompare 插件不是万能的银弹,不过在遇到如下情况时,它确实能帮助你提升应用的性能:

- 需要应用浅度比较的组件是“纯粹的”(换句话说,在使用相同的 props 和 state 时,渲染结果总是相同的)。
- 你使用了不可变值,或使用了 React 中的不变性助手来处理状态。

### 看板应用:使用 shallowCompare 插件实现 shouldComponentUpdate

看板应用整体上有着不错的性能,不过有一个地方会有些卡顿:拖曳卡片时。这是因为每次你改变卡片位置或所属列表时,所有卡片都会重新进行渲染。为了解决这个问题,下面在 Card 组件中实现 shouldComponentUpdate 生命周期方法,同时使用 shallowCompare 插件。

作为开始,下面在 npm 中安装 react-addons-shallow-compare:

```
npm install --save react-addons-shallow-compare
```

接下来编辑 Card 组件,导入 shallowCompare 并实现 shouldComponentUpdate 方法,如代码清单 7-9 所示。



代码清单 7-9: 在 Card 组件中实现 shouldComponentUpdate

```

import React, { Component, PropTypes } from 'react';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import marked from 'marked';
import { DragSource, DropTarget } from 'react-dnd';
import constants from '../constants';
import CheckList from './CheckList';
import { Link } from 'react-router';
import CardsActionCreators from '../actions/CardsActionCreators';
import shallowCompare from 'react-addons-shallow-compare';

let titlePropType = (props, propName, componentName) => {...};
const cardDragSpec = {...};
const cardDropSpec = {...};
let collectDrag = (connect, monitor) => {...};
let collectDrop = (connect, monitor) => {...};

class Card extends Component {
  toggleDetails() {...}

  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState)
  }

  render() {...}
};
Card.propTypes = {...};

const dragHighOrderCard = DragSource(constants.CARD, cardDragSpec,
  collectDrag)(Card);
const dragDropHighOrderCard = DropTarget(constants.CARD, cardDropSpec,
  collectDrop)(dragHighOrderCard);
export default dragDropHighOrderCard;

```

## 7.4 本章小结

在本章中, 你对在 React 中如何快速实现子级校正算法有了更好的理解。如你所见, 虽然在绝大多数情况下它已经足够快了, 但我们仍有能力手动提升组件的性能: 通过实现 shouldComponentUpdate 生命周期方法来阻止组件(及其整个子树)的重绘。



## React 同构应用

简单讲，单页应用基本上就是一个空白的 HTML 体，使用 JavaScript 来生成应用的页面。这种方式虽然有很多优势，但也容易看到其中存在不利的一面：在浏览器下载应用的 JavaScript 并运行(以及从服务器上获取最初的数据)之前，用户会看到一个白屏闪过，然后才是页面的内容。

同构(isomorphic)JavaScript 应用也称为通用 JavaScript 应用，指的是在客户端和服务端之间完整(或部分)地共享代码的应用。通过在服务器端运行应用的 JavaScript 代码，页面可在发送到浏览器之前预先填充内容，所以用户甚至可在浏览器的 JavaScript 运行之前就先看到内容。当本地的 JavaScript 运行时，它会接手后续的交互及导航操作，通过快速的初始化加载和服务器端页面渲染，让用户在单页应用程序中得到流畅的交互体验。

在这个过程中，用户因为应用加载和渲染的速度更快有了更好的体验，同时还获得了其他好处：获得了渐进式增强(progressive enhancement)能力(在 JS 加载失败时应用也不会完全停止工作)、更好的可访问性，并使搜索引擎更容易通过索引搜索应用内容。

### 8.1 Node.js 和 Express

为使 React 应用能在服务器上运行和预填充，需要使用 Node.js 和 Express。从第 1 章开始，你就已经用过 Node.js 和 Node 的包管理器(npm)。正如你之前看到的那样，Node.js 是一个 JavaScript 运行时，能在浏览器之外执行 JavaScript 应用。虽然它已成为用于本地开发和客户端 JavaScript 项目包管理的一个重要工具，它确实也能在服务器端 JavaScript 解决方案(特别是在构建类似 Web 服务器之类的网络应用时)闪耀光芒，它的功能类似于 PHP 和 Python。

Express 是 Node.js 中的一个 Web 应用服务器框架，用于构架单页、多页或混合 Web 应用。它运用广泛，已成为 Node.js 中的事实服务器框架。

下一节对 Express 进行简要介绍，其中不涉及与 React 相关的内容。如果你已经对 Express 非常熟悉，尽管跳过这一节吧。

## 使用 Node.js 和 Express 编写“Hello World”

讲解 Node.js 和 Express 已经超出了本书范畴,不过为让读者熟悉开发 React 同构应用之前所需的设置,下面创建一个简单的 Node.js 和 Express 的 HelloWorld 应用。从一个新的空白文件夹开始,首先创建一个 package.json 项目文件(使用 `npm init -y` 命令,并接受所有默认设置)和一个 server.js 文件。

接下来,安装项目的依赖项: Express 框架和 Babel(能够让你使用最新 JavaScript 特性的编译器)。使用 `npm install --save express` 来安装 Express。

Babel 的安装过程则稍微复杂一些。默认情况下, babel-core 这个包并不处理任何事。为满足我们代码中的需要,需要启用插件(或者插件的组合,也就是预配置)。在这个示例中,你会使用 ES6 的预配置: `npm install --save babel-core babel-preset-es2015`。

最后还需要安装 babel-cli 包,用于在命令行中编译文件。可使用 `npm install --global babel-cli` 在全局安装 Babel 的命令行编译工具。

### 1. 配置 Babel

为使 Babel 正常工作,需要根据项目来进行配置。配置 Babel 最简单的方法是在项目的根目录中创建一个 .babelrc 文件。在这个示例中,配置中只需要包含 ES6 预设。代码清单 8-1 展示了最终的 .babelrc 配置文件。

#### 代码清单 8-1: .babelrc 配置文件

```
{
  "presets": ["es2015"]
}
```

### 2. 创建 Express 服务器

创建一个新的 server.js 文件,开始编写你的服务器端应用。到目前为止,项目结构应该如图 8-1 所示。

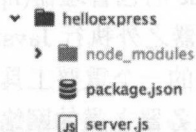


图 8-1 Node.js 和 Express 项目的基本骨架

在 server.js 文件中,需要 Express 框架,并创建一个 Express 服务器实例。为方便起见,使用 app 常量指向 express.Server,如下所示:

```
import express from 'express';
const app = express();
```

接下来,为你的应用设置一个或多个路由。路由是由路径(字符串或正则表达式)、回调函数、HTTP 方法组成的。回调函数接收两个参数: `request` 和 `response`。`request` 对象包含了触发这个事件的 HTTP 请求信息(包括查询字符串、参数、`body`、HTTP 头等)。为响应这个请求,使用 `response` 对象将所需的 HTTP 响应内容发回到浏览器端。

Hello World 示例调用了 `app.get()` 方法,用来表示 HTTPGET 方法,使用路径 `"/"`,在回调函数中将一个字符串返回给浏览器:

```
app.get('/', (request, response) =>{
  response.send('<html><body><p>Hello World!</p></body></html>');
});
```

最后,可让服务器在指定端口上开始侦听。在下面的代码中,调用了 `listen()` 方法,指定了 3000 端口和一个回调方法(在服务器开始运行时调用):

```
app.listen(3000, ()=>{
  console.log('Express app listening on port 3000');
});
```

`server.js` 完整的源代码如代码清单 8-2 所示。

#### 代码清单 8-2: `server.js` 源代码

```
import express from 'express';
const app = express();

app.get('/', (request, response) =>{
  response.send('<html><body><p>Hello World!</p></body></html>');
});

app.listen(3000, ()=>{
  console.log('Express app listening on port 3000');
});
```

### 3. 运行服务器

在终端中输入如下命令,以调试模式(可看到由 Express 生成的日志)启动服务器:

```
DEBUG=express:* babel-node server.js
```

在服务器运行时,可以通过你的浏览器来访问 `localhost:3000`。结果类似图 8-2 所示。为进行简化和少打些字,可在 `package.json` 中传入这个命令作为启动脚本。这样,下一次需要启动本地服务器时,只需要输入 `npm start` 即可。代码清单 8-3 展示了更新后的 `package.json` 文件。

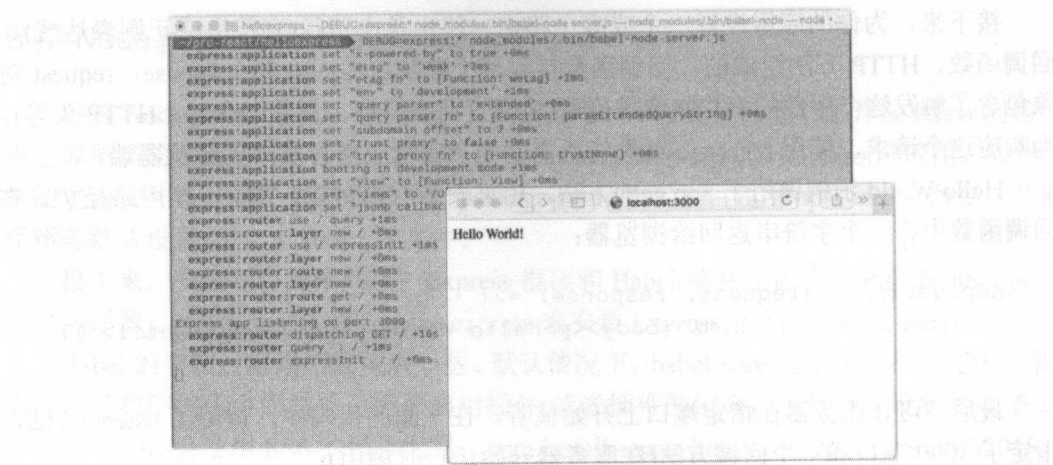


图 8-2 运行 Node.js 和 Express 服务器，并通过浏览器进行测试

代码清单 8-3: 更新后的 package.json

```
{
  "name": "helloexpress",
  "version": "0.0.1",
  "description": "Hello world sample application in Node.js + Express",
  "scripts": {
    "start": "DEBUG=express:* babel-node server.js"
  },
  "author": "Cássio Zen",
  "license": "ISC",
  "dependencies": {
    "babel": "^5.8.29",
    "express": "^4.13.3"
  }
}
```

4. 使用模板

使用response.send发送字符串内容的响应，是用来快速上手Express的，不过在实际场景中通过这种方法来格式化输出所有响应、输出完整的HTML结构，那就太麻烦了。为此，Express支持使用模板。模板是一组HTML标记，其中带有一些标签用于插入变量或运行程序逻辑。Express支持大量的模板格式，在这个示例中，我们使用EJS模板格式。首先，通过 npm install --global ejs 来安装 EJS 作为应用的依赖项。接下来，为使用 EJS 模板，需要对应用进行配置。可以使用 set 方法来完成：

```
app.set('view engine', 'ejs');
```

默认情况下，模板文件必须保存在 views 文件夹中。创建一个新的 views 文件夹，和一个 index.ejs 模板文件，如图 8-3 所示。



图 8-3 模板文件夹和 index.ejs 模板文件

为了让应用渲染模板，而非发送字符串，需要使用 `response.render` 方法，传入模板名称和用于在模板内显示动态值的对象。代码清单 8-4 中展示了完整的 `server.js` 源代码，其中包含了使用模板进行渲染的内容。代码清单 8-5 展示了 `index.ejs` 源代码。

#### 代码清单 8-4: 更新后的 `server.js`，使用模板进行渲染

```
import express from 'express';
const app = express();

app.set('view engine', 'ejs');

app.get('/', (request, response) => {
  response.render('index', {message: 'Hello World'});
});

app.listen(3000, () => {
  console.log('Express app listening on port 3000');
});
```

#### 代码清单 8-5: `views/index.ejs` 模板文件

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Express Template</title>
  </head>
  <body>
    <h1><%= message %></h1>
  </body>
</html>
```

### 5. 静态资源访问

Express 内置中间件来提供静态资源的访问。`express.static()` 中间件接收一个参数，指向静态资源所在文件的根目录。从 `public` 文件夹提供静态资源访问，在服务器端代码中添加如下行：

```
app.use(express.static(__dirname + '/public'));
```



## 8.2 React 同构基础

现在你已经了解了 Express，下面更进一步，看看如何在服务器端渲染一个实际的 React 组件吧。React 和 React-DOM 包通过 ReactDOMServer.renderToString 方法内置支持在服务器端渲染组件的能力，该方法会对所需组件进行渲染，并生成带有注释的 HTML 然后发送到浏览器。在浏览器中，React 会识别这些注释，并只进行事件处理程序的加载，从而使得应用在初次加载时获得极佳性能。

### 8.2.1 创建项目结构

与纯客户端应用相比，React 同构应用项目结构会有些不同的需求。因此，我们不再使用之前的 React App Boilerplate 应用作为新项目的模板，下面从头开始创建一个新的项目结构。

在一个新的文件夹中，第一件要做的事就是创建 package.json 项目文件。你可以通过 npm init -y 来快速完成该文件的创建。在项目的根目录中，会包含两个文件夹：public 文件夹(用于存放浏览器中使用的静态资源)和 app 文件夹(用于保存 React 组件，以及那些在服务器端和客户端之间共享的项目文件)。在 app 文件夹中，再创建一个 components 文件夹来组织你的项目内容。

项目从 3 个文件开始：server.js、index.ejs 和 browser.js。server.js 文件中会包含服务器端的 JavaScript 代码(用于配置 Express 服务器并渲染组件)。browser.js 文件中会包含客户端 JavaScript 代码。而 index.ejs 文件则会包含基本的 HTML 页面结构，它会被发送到浏览器中。图 8-4 展示了项目结构中的文件和文件夹。



图 8-4 项目结构

#### 联系人应用文件

在这个项目中，你将使用类似第 3 章中的示例：一个带有搜索栏的联系人列表组件。不过项目文件不会完全一样：你将创建一个简化的组件层级结构，把主要关注点放在服务器端渲染上；通过名为 initialData 的属性接收联系人数组。组件层级结构包括：



- **ContactsApp**: 主体组件
  - **SearchBar**: 显示一个 input 域, 用户可以使用它对联系人进行筛选
  - **ContactList**: 遍历数据, 并创建一组 **ContactItems**

图 8-5 展示了我们期望的输出结果。

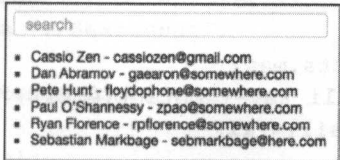


图 8-5 联系人应用

总之, React 项目包含 4 个文件: 3 个组件和 1 个包含联系人代码清单的 JSON 文件。我们在 `components` 文件夹中创建所有组件相关的文件, 源代码包含在代码清单 8-6 至代码清单 8-8 中。而 JSON 文件则保存在 `public` 文件夹中, 其内容见图 8-6。

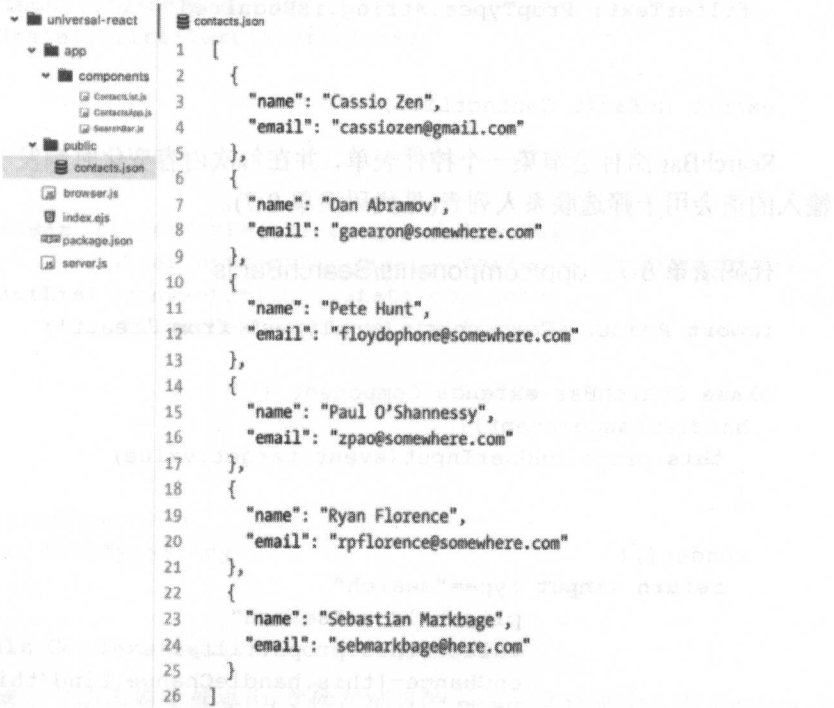


图 8-6 项目根文件夹中的 `contacts.json` 文件

`ContactList` 组件接收一个联系人数组和一个 `filterText` 属性, 它会对联系人进行筛选, 然后遍历数组, 依次渲染每个联系人的信息(见代码清单 8-6)。

代码清单 8-6: `app/components/ContactList.js`

```
import React, {Component, PropTypes} from 'react';
```

```

class ContactList extends Component {
  render() {
    var filteredContacts = this.props.contacts.filter(
      (contact) => contact.name.indexOf(this.props.filterText) !== -1
    );
    return (
      <ul>
        {filteredContacts.map(
          (contact) => <li key={contact.email}>{contact.name} -
            {contact.email}</li>
        )}
      </ul>
    )
  }
}

ContactList.propTypes = {
  contacts: PropTypes.arrayOf(PropTypes.object),
  filterText: PropTypes.string.isRequired
}

export default ContactList;

```

SearchBar 组件会渲染一个控件表单，并在每次内容变化时触发一个回调函数。用户输入的值会用于筛选联系人列表(见代码清单 8-7)。

#### 代码清单 8-7: app/components/SearchBar.js

```

import React, {Component, PropTypes} from 'react';

class SearchBar extends Component {
  handleChange(event) {
    this.props.onUserInput(event.target.value)
  }

  render() {
    return <input type="search"
      placeholder="search"
      value={this.props.filterText}
      onChange={this.handleChange.bind(this)} />
  }
}

SearchBar.propTypes = {
  onUserInput: PropTypes.func.isRequired,
  filterText: PropTypes.string.isRequired
}

export default SearchBar;

```

ContactApp 用来渲染 ContactList 组件和 SearchBar 组件。它会通过 initialData 属性

来接收初始的联系人列表，并将该属性作为控件自身的状态(见代码清单 8-8)。

代码清单 8-8: app/components/ContactsApp.js

```
import React, {Component, PropTypes} from 'react';
import ContactList from '../ContactList';
import SearchBar from '../SearchBar';

class ContactsApp extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      contacts: this.props.initialData || [],
      filterText: ''
    }
  }

  handleUserInput(searchTerm) {
    this.setState({filterText: searchTerm})
  }

  render() {
    return (
      <div>
        <SearchBar filterText={this.state.filterText}
          onUserInput={this.handleUserInput.bind(this)} />
        <ContactList contacts={this.state.contacts}
          filterText={this.state.filterText}/>
      </div>
    )
  }
};

ContactsApp.propTypes = {
  initialData: PropTypes.any
};

export default ContactsApp;
```

最后，图 8-6 展示了应用根文件夹中(文件在项目的 public 文件夹中，而 Express 会在访问时针对静态内容将其映射为根文件夹)的 contacts.json 文件。

## 8.2.2 在服务器端渲染 React 组件

到目前为止，应用的文件夹结构和示例组件都已经写好，现在可以开始编写服务器端脚本了。作为开始，首先来安装服务器端开发的依赖项：Express 服务器、EJS 模板格式以及 React 包。通过 `npm install --save express ejs react react-dom` 命令来一次性安装这些内容。

此外还需要安装Babel,这样就可以使用ES6语法和JSX。要安装带有ES6和JSX支持的最新版的Babel及其依赖项(webpack),使用命令: `npm install --save webpack babel-core babel-loader babel-preset-es2015 babel-preset-react`。

最后,还需要安装Babel的命令行编译工具。为便于后续使用,可使用命令 `npm install --global babel-cli` 把它安装到全局环境中。

## 1. 配置 Babel

为能让Babel正常工作,需要针对项目对其进行配置。配置Babel最简单的方法是在项目的根文件夹中创建一个**.babelrc**文件。在我们的项目中,配置文件包含了最简单的ES6预设配置。代码清单8-9展示了最终的**.babelrc**配置文件。

代码清单 8-9: .babelrc 配置文件

```
{
  "presets": ["es2015", "react"]
}
```

## 2. Express 应用

模板文件非常简单:使用基本的HTML标签,包含一个root div,用于插入动态内容。代码清单8-10展示了其源代码。

代码清单 8-10: index.ejs 模板文件

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Isomorphic React</title>
  </head>
  <body>
    <div id="root"><%- content %></div>
  </body>
</html>
```

接下来,在server.js文件中设置Express服务器,需要考虑如下几点:

- Express服务器做如下配置:
  - 使用EJS作为模板格式,并配置为在根目录中查找模板文件
  - 通过public文件夹提供静态资源的访问
- ContactApp组件需要一个联系人列表。这个列表可来自数据库或API服务器,不过为简化场景,我们将加载public文件夹中存放的JSON文件。

服务器端代码如代码清单8-11所示。

## 代码清单 8-11: 带有联系人列表的基本 Express 应用

```
import express from 'express';
import contacts from './public/contacts.json';

const app = express();

app.set('views', './')
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));
app.get('/', (request, response) =>{
  response.render('index', {
    content: 'Hello'
  });
});

app.listen(3000, ()=>{
  console.log('Express app listening on port 3000');
});
```

所有事情都已经准备就绪, 你现在可以做一下测试, 不过结果只是在浏览器中会输出一个 Hello 字符串。使用 `node_modules/.bin/babel-node server.js` 来启动服务器。

## 3. 渲染 React 组件

接下来就是有趣的部分了: 在服务器端渲染 React 组件。这里面需要考虑如下问题:

- 前面提到过, 我们会使用 `react-dom` 中的一个 `renderToString` 方法, 根据组件来生成带有注解的标记语言, 然后发送到浏览器中。
- 在 Express 服务器中不会使用 JSX。正如我们在第 2 章中提到的, 为在 JSX 之外实例化 React 组件, 我们需要在调用之前使用工厂方法对组件进行封装。

代码清单 8-12 展示了更新后的服务器端代码。

## 代码清单 8-12: 渲染 React 组件的 Express 应用

```
import fs from 'fs';
import express from 'express';
import React from 'react';
import { renderToString } from 'react-dom/server';
import ContactsApp from './app/components/ContactsApp';

const app = express();
app.set('views', './')
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

const contacts = JSON.parse(fs.readFileSync(__dirname + '/public/contacts.json', 'utf8'));
```

```
const ContactsAppFactory =React.createFactory(ContactsApp);

app.get('/', (request, response) =>{
  let componentInstance =ContactsAppFactory({initialData:contacts});
  response.render('index',{
    content:renderToString(componentInstance)
  });
});

app.listen(3000, ()=>{
  console.log('Express app listening on port 3000');
});
```

如果你启动服务器，然后在浏览器中进行测试，会看到组件已经正确渲染，生成的 HTML 中包括了一些必要的注释，客户端的 React 可以通过这些注释来挂载组件。结果应如图 8-7 所示。

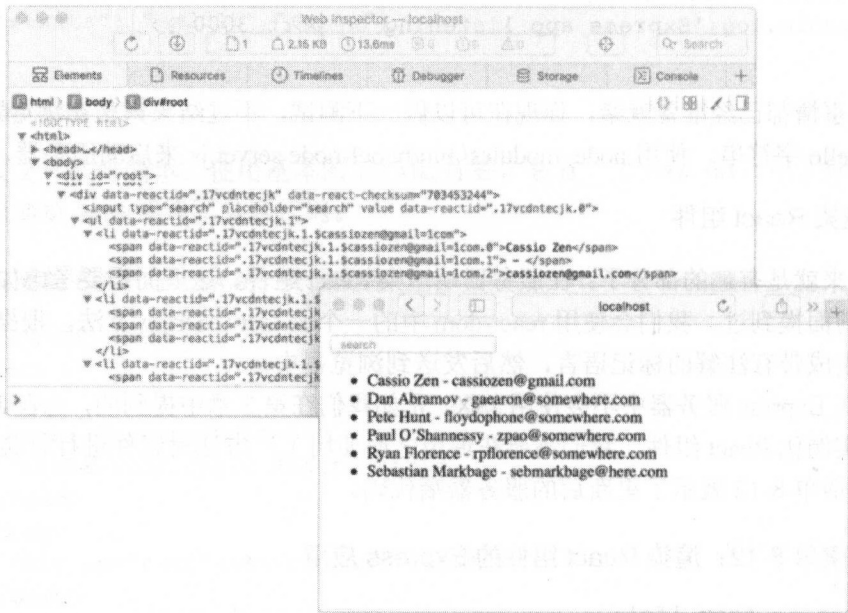


图 8-7 服务器端渲染的 React 组件

不过需要注意，到目前为止，在浏览器中是没有交互操作的。这个组件是静态的，并没有对联系人进行筛选。这是因为你并没有创建或发送浏览器中需要运行的 JavaScript 文件，你只创建了服务器端的 JavaScript 脚本。浏览器只是接收并显示了 HTML 页面，其中不包含任何动态内容。

8.2.3 在客户端中挂载 React

我们目前在创建一个 React 同构应用项目，可用同样的组件代码在服务器和客户端中渲染组件，不过目前为止我们还没有完成，因为现在只在服务器端进行了渲染。我们



还需要提供一个 JavaScript 文件,能让浏览器在预先渲染好的页面中挂载 React 组件,并加载事件侦听器。

## 1. 客户端配置

目前我们已经有一个空的 JavaScript 文件(browser.js),用来生成客户端的 JavaScript 文件,不过这个文件在发送到浏览器之前,需要进行编译和打包。

在本书的所有示例中,我们都用 webpack 和 Babel 来完成这一过程,这里其实也没什么不同。在根文件夹中创建一个 webpack 配置文件,它很简单,因为我们并不需要高级功能(例如 webpack 的开发服务器),只是单纯将 browser.js 文件打包,然后输出一个 bundle.js 文件到 public 文件夹中,如代码清单 8-13 所示。

代码清单 8-13: webpack.config.js 文件

```
module.exports={
  entry: [
    './browser.js'
  ],
  output:{
    path:'./public',
    filename:"bundle.js"
  },
  module:{
    loaders: [{
      test:/\.jsx?$/,
      loader:'babel'
    }]
  }
};
```

编辑好这个配置文件后,可运行 webpack -p 来生成打包好的客户端 JavaScript 文件。

## 2. 传递组件的初始数据(initialData 属性)

现在,我们已经有了一个配置文件,能将 browser.js 文件编译并打包成 public 文件夹中的 bundle.js 文件,我们需要对模板文件进行一些更新,来加载这个打包后的 JavaScript 文件。

当然,我们需要更新的不仅是模板文件。挂载服务器端渲染好的组件和客户端渲染都不一样,我们需要使用和服务端渲染组件时完全相同的属性,否则,React 会强制重新渲染整个 DOM(同时 React 会给出一个警告信息)。这里,我们需要一个机制来传递和服务端同样的属性给客户端。为此,我们可在 HTML 模板中创建一个 script 标签,然后在其中转储所有属性。然后客户端的 JavaScript 就可以对其进行解析,并使用这些完全相同的属性。

通俗地讲,我们在模板文件中需要两个 script 标签:一个包含 React 组件需要使用的初始数据(所有数据和属性),另一个用来加载客户端 JavaScript 文件。代码清单 8-14 展

示了更新后的 index.ejs 文件。

代码清单 8-14: 更新后的 index.ejs 模板文件, 带有两个 script 标签

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Isomorphic React</title>
  </head>
  <body>
    <div id="root"><%- content %></div>

    <script id="initial-data" type="application/json">
      <%-reactInitialData%>
    </script>
    <script type="text/JavaScript" src="bundle.js"></script>
  </body>
</html>
```

代码清单 8-15 展示了更新后的 server.js, 更新了 render 方法, 传递了联系人数组, 客户端用它对 ContactsApp 组件进行本地初始化。

代码清单 8-15: 更新后的 server.js 文件, 在 script 标签中生成 JSON 格式的联系  
人列表

```
import fs from 'fs';
import express from 'express';
import React from 'react';
import { renderToString } from 'react-dom/server';
import ContactsApp from './app/components/ContactsApp';

const app = express();
app.set('views', './');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

const contacts = JSON.parse(fs.readFileSync(__dirname + '/public/contacts.json', 'utf8'));

const ContactsAppFactory = React.createFactory(ContactsApp);

app.get('/', (request, response) =>{
  let componentInstance = ContactsAppFactory({contacts:contacts});
  response.render('index',{
    reactInitialData:JSON.stringify(contacts),
    content:renderToString(componentInstance)
  });
});
```

```
app.listen(3000, ()=>{
  console.log('Express app listening on port 3000');
});
```

### browser.js

browser.js 和 server.js 文件类似：导入所需的组件，然后渲染它。我们还需要传递和服务端渲染时完全相同的属性。如果你传入的属性不同，客户端的 React 就无法在预先渲染好的组件上直接进行“挂载”、分配事件侦听器以及处理交互操作。幸运的是，服务器端已在一个 id 为 initialData 的 script 标签中包含了这个属性，客户端 JavaScript 可以对其进行解析并直接使用。

代码清单 8-16 展示了完整的代码。

### 代码清单 8-16: browser.js 文件

```
import React from 'react';
import { render } from 'react-dom';
import ContactsApp from './app/components/ContactsApp';

let initialData = document.getElementById('initial-data').textContent;
if(initialData.length>0){
  initialData = JSON.parse(initialData);
}

render(<ContactsApp initialData={initialData} />,
  document.getElementById('root'));
```

为避免 initialData 不存在时造成的错误，注意我们在解析之前检查了 script 标签中的初始数据是否有内容。

启动服务器开始测试之前，确认已使用 webpack -p 命令编译并打包了 browser.js 文件，并输出到 public/bundle.js 中。见图 8-8。

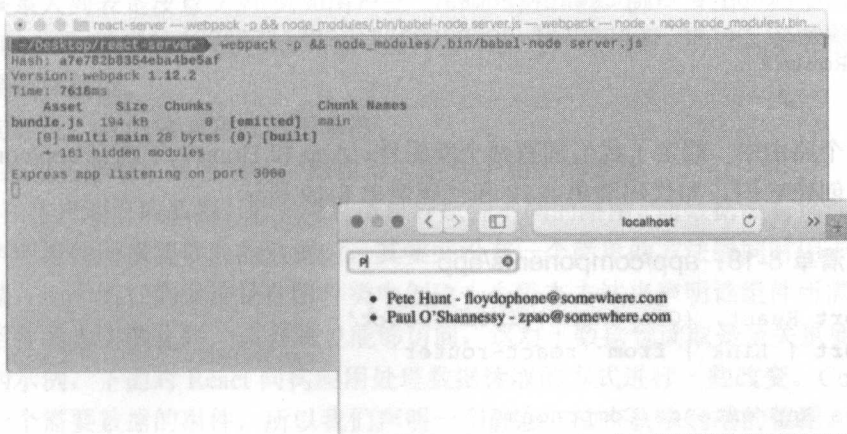


图 8-8 客户端的 React 在服务器端预先生成的内容基础之上挂载组件

## 8.3 路由

React Router 已成为 React 应用中路由解决方案的既成标准，从它的 1.0 版本发布时起，就已经内置了服务器端渲染支持。不过在服务器端配置路由和在客户端略有不同，除了匹配路由和组件之外，还需要针对错误发送 500 响应，针对重定向发送 30x 响应。

为帮助理解这些需求，我们从 <Router> API 向下一级：

- 使用 Match 将路由匹配到一个位置，而不需要进行渲染
- RoutingContext 用于路由组件的同步渲染

### 8.3.1 配置内部路由

我们会继续在 ContactsApp 示例上继续改造，我们将实现一个名为 Home 的新路由。首先安装 React Router。注意本书中的示例使用了 React Router 1.0.0。我们同样还需要它的依赖项 history。使用 `npm install --save react-router history` 来安装这些内容。

接下来，对路由文件进行配置。因为路由将在服务器端和客户端之间共享，我们在 app 文件夹中保存该文件。先建立 3 个路由：父级的 App 路由、Home 首页路由和用于展示现有 ContactsApp 的 Contacts 路由。代码清单 8-17 展示了源代码。

代码清单 8-17: app/routes.js

```
import React from 'react';
import { Route, IndexRoute } from 'react-router'
import App from './components/App'
import Home from './components/Home'
import ContactsApp from './components/ContactsApp'

export default (
  <Route path="/" component={App}>
    <IndexRoute component={Home} />
    <Route path="contacts" component={ContactsApp} />
  </Route>
);
```

在这个路由中，假定我们拥有两个新组件：App 和 Home。下面在 app/components 文件夹中创建它们，如代码清单 8-18 和代码清单 8-19 所示。

代码清单 8-18: app/components/app

```
import React, {Component} from 'react';
import { Link } from 'react-router'

class App extends Component {
  render() {
    return(
```

```

    <div>
      <nav>
        <Link to="/">Home</Link>{' '}
        <Link to="/contacts">Contacts</Link>
      </nav>
      <div>
        {this.props.children}
      </div>
    </div>
  )
}
};

export default App;

```

代码清单 8-19: app/components/home 文件

```

import React, {Component} from 'react';

class Home extends Component {
  render() {
    return <h1>Home</h1>;
  }
};

export default Home;

```

### 8.3.2 动态数据获取

到目前为止，我们的项目只有一个入口点：ContactsApp 组件。每次在浏览器中加载应用时，服务器端都会预先加载联系人列表，然后将一个预填充的组件发送到浏览器中。在拥有多个路由的应用中，服务器需要检测到当前路由映射的组件需要哪些数据，始终都加载联系人列表是没意义的(例如用户在 Home 路由时)。另一方面，如果用户在一开始导航到 Home 然后进入 Contact 路由，组件并不会进行预填充，所以有必要根据情况从服务器端获取数据。

换句话说，我们需要完成如下功能：

- 在服务器端，只预读当前路由对应组件中相关的数据
- 在客户端获取数据，用于应对用户导航到不同的路由，而数据没有预读取的情况

在声明组件所需要获取的数据时，其实并没有一个简单的方法能同时用于服务器端和客户端，一个流行的做法是在组件类中创建一个静态方法来声明该组件所需的数据。即使在组件尚未实例化时，该方法也能够访问，这对于数据预读取是至关重要的。

作为示例，下面对 React 同构应用处理数据读取的方式进行一些改变。ContactsApp 是唯一一个需要数据的组件，所以我们声明一个静态方法来获取远端的数据。如果用户通过 Contacts 路由进入了应用，服务器端会运行这个方法，并将结果传递给组件的属性。

如果用户通过其他路由进入应用，随后导航到 `Contacts` 路由，浏览器会运行这个方法，根据需要来获取数据。

首先安装 `isomorphic-fetch` 这个 `npm` 包。这个包智能地在服务器端使用了 `Node` 默认的获取方法，同时在浏览器端实现了一个替补：`npm install -save isomorphic-fetch`。

接下来更新 `ContactsApp` 组件来创建一个静态的 `requestInitialData` 方法。

我们同样还需要实现 `componentDidMount` 生命周期方法，它只会在浏览器端被调用。该方法检查 `initialData` 是否已经通过服务器端提供，如果没有的话，则立即调用 `requestInitialData` 方法来获取初始化数据。代码清单 8-20 展示了更新后的代码。

#### 代码清单 8-20：更新后的组件

```
import React, {Component, PropTypes} from 'react';
import fetch from 'isomorphic-fetch';
import ContactList from './ContactList';
import SearchBar from './SearchBar';

class ContactsApp extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      contacts: this.props.initialData || [],
      filterText: ''
    }
  }

  componentDidMount() {
    if (!this.props.initialData) {
      ContactsApp.requestInitialData().then(contacts => {
        this.setState({ contacts });
      });
    }
  }

  handleUserInput(searchTerm) {
    this.setState({filterText:searchTerm})
  }

  render() {
    return (
      <div>
        <SearchBar filterText={this.state.filterText}
          onUserInput={this.handleUserInput.bind(this)} />
        <ContactList contacts={this.props.initialData}
          filterText={this.state.filterText}/>
      </div>
    )
  }
}
```



```

};

ContactsApp.propTypes = {
  initialData: PropTypes.any
};

ContactsApp.requestInitialData = () => {
  return fetch('http://localhost:3000/contacts.json')
    .then((response) => response.json());
};

export default ContactsApp;

```

### 8.3.3 渲染路由

为同时在客户端和服务端进行路由的渲染，我们需要对 `server.js` 和 `browser.js` 进行大幅改动。

#### 1. 在服务器端渲染路由

从 `server.js` 开始，一步一步进行修改。首先，把 `get` 入口从 `“/”` 修改为 `“*”`，这样所有路由都会触发回调函数。同样需要对错误、未找到页面、路由重定向进行配置。对于已经存在的路由，渲染相应的组件。代码清单 8-21 展示了这些变化。

代码清单 8-21：对 `server.js` 的第一步修改

```

import fs from 'fs';
import express from 'express';
import React from 'react';
import { renderToString } from 'react-dom/server';
import { match, RoutingContext } from 'react-router';
import routes from './app/routes';

const app = express();

app.set('views', './');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

const contacts = JSON.parse(fs.readFileSync(__dirname + '/public/contacts.json', 'utf8'));

let renderRoute = (response, renderProps) => {
  // The actual rendering will be moved here
};

app.get('*', (request, response) => {
  match({ routes, location: request.url }, (error, redirectLocation,

```

```

renderProps) =>{
  if (error) {
    response.status(500).send(error.message);
  }else if (redirectLocation) {
    response.redirect(302,redirectLocation.pathname+
      redirectLocation.search);
  }else if (renderProps) {
    renderRoute(response, renderProps);
  }else{
    response.status(404).send('Not found');
  }
});
});
app.listen(3000, ()=>{...});

```

上述代码中有一些值得注意的地方：

- `importContactApp` 和工厂方法被移除了(因为这些工作现在由路由来完成)。
- 为更好地组织代码，把 `response.render` 方法移到路由之外，形成一个新函数 `renderRoute`(会在后续步骤实现它)。

下一步，重新实现组件的渲染。`React Router` 提供了一个名为 `RoutingContext` 的对象，其中带有需要渲染到当前路由中的所有组件的层级结构。我们可以直接把 `RoutingContext` 传递给 `React` 的 `renderToString` 方法，来为所有组件生成标记语言，不过在我们的场景中这个方法不适用，因为我们还需要通过 `requestInitialData` 静态方法对数据进行预读取，然后将其作为属性传递给组件。

我们需要遍历 `RoutingContext` 中的所有组件，检查它们是否实现了 `requestInitialData` 方法。如果你找到一个组件实现了该方法，就通过重写 `RoutingContext` 中用于实例化内部组件的方法，预读取所需的数据并将其作为属性传递给内部组件。代码清单 8-22 展示了更新后的代码。

代码清单 8-22：对 `server.js` 的第二步修改

```

import ...;

const app =express();
app.set(...);
app.set(...);
app.use(...);
const contacts =JSON.parse(...);

// Helper function: Loop through all components in the renderProps object
// and returns a new object with the desired key
let getPropsFromRoute = ({routes}, componentProps) =>{
  let props ={};
  let lastRoute = routes[routes.length-1];
  routes.reduceRight((prevRoute, currRoute) =>{

```

```

    componentProps.forEach(componentProp =>{
      if (!props[componentProp] && currRoute.component[componentProp]) {
        props[componentProp] =currRoute.component[componentProp];
      }
    });
  }, lastRoute);
  return props;
};

let renderRoute = (response, renderProps) =>{
  // Loop through renderProps object looking for 'requestInitialData'
  let routeProps =getPropsFromRoute(renderProps,
    ['requestInitialData']);
  if (routeProps.requestInitialData) {
    // If one of the components implements 'requestInitialData', invoke it.
    routeProps.requestInitialData().then((data)=>{
      // Ovwewrite the react-router create element function
      // and pass the pre-fetched data as initialData props
      let handleCreateElement = (Component, props) =>{
        <Component initialData={data}{...props}/>
      };
      // Render the template with RoutingContext and loaded data.
      response.render('index',{
        reactInitialData: JSON.stringify(data),
        content: renderToString(
          <RoutingContext createElement={handleCreateElement}
            {...renderProps} />
        )
      });
    });
  } else{
    // No components in this route implements 'requestInitialData'.
    // Simply render the template with RoutingContext and no initialData.
    response.render('index',{
      reactInitialData:null,
      content:renderToString(<RoutingContext {...renderProps}/>)
    });
  }
};

app.get('*', (request, response) => {...});
app.listen(3000, ()=>{...});

```

更新后的 server.js 文件的完整代码如代码清单 8-23 所示。

代码清单 8-23: server.js 文件更新后的完整代码

```

import fs from 'fs';
import express from 'express';
import React from 'react';

```

```

import { renderToString } from 'react-dom/server';
import { match, RoutingContext } from 'react-router';
import routes from './app/routes';
const app = express();
app.set('views', './');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

const contacts = JSON.parse(fs.readFileSync(__dirname + '/public/
  contacts.json', 'utf8'));

let getPropsFromRoute = ({ routes }, componentProps) => {
  let props = {};
  let lastRoute = routes[routes.length - 1];
  routes.reduceRight((prevRoute, currRoute) => {
    componentProps.forEach(componentProp => {
      if (!props[componentProp] && currRoute.component[componentProp]) {
        props[componentProp] = currRoute.component[componentProp];
      }
    });
  }, lastRoute);
  return props;
};

let renderRoute = (response, renderProps) => {
  let routeProps = getPropsFromRoute(renderProps,
    ['requestInitialData']);
  if (routeProps.requestInitialData) {
    routeProps.requestInitialData().then((data) => {
      let handleCreateElement = (Component, props) => {
        <Component initialData={data} {...props}/>
      };
      response.render('index', {
        reactInitialData: JSON.stringify(data),
        content: renderToString(
          <RoutingContext createElement={handleCreateElement}
            {...renderProps} />
        )
      });
    });
  } else {
    response.render('index', {
      reactInitialData: null,
      content: renderToString(<RoutingContext {...renderProps}/>)
    });
  }
};

app.get('*', (request, response) => {

```

```

match({ routes, location: request.url }, (error, redirectLocation,
  renderProps) => {
  if (error) {
    response.status(500).send(error.message);
  } else if (redirectLocation) {
    response.redirect(302, redirectLocation.pathname +
      redirectLocation.search);
  } else if (renderProps) {
    renderRoute(response, renderProps);
  } else {
    response.status(404).send('Not found');
  }
});
});

app.listen(3000, ()=>{
  console.log('Express app listening on port 3000');
});

```

## 2. 在浏览器中渲染路由

接下来，需要在 `browser.js` 中做类似的调整：渲染路由并检查是否存在 `initialData`，将其作为属性传递给正确的组件。同样，需要依赖 `React Router` 的 `createElement` 属性来重写实例化 `React` 元素的默认方法，将 `initialData` 作为属性传递给实现了 `requestInitialData` 静态方法的组件。代码清单 8-24 展示了更新后的代码。

代码清单 8-24：更新后的 `browser.js` 代码

```

import React from 'react';
import { render } from 'react-dom';
import { Router } from 'react-router';
import { createHistory } from 'history';
import routes from './app/routes';

let handleCreateElement = (Component, props) => {
  if(Component.hasOwnProperty('requestInitialData')){
    let initialData = document.getElementById(
      'initial-data').textContent;
    if(initialData.length>0){
      initialData = JSON.parse(initialData);
    }
    return <Component initialData={initialData} {...props} />;
  } else {
    return <Component {...props} />;
  }
}

render((
  <Router history={createHistory()} createElement={handleCreateElement}>

```

```
{routes}</Router>), document.getElementById('root'))
```

如果现在访问 Home 路由(“/”), 就会注意到当访问联系人路由时, 该组件会通过浏览器来获取数据。然而如果你直接访问“/contacts”路由, 服务器端会预读取联系人数据, 并将一个已填充好的组件发送到浏览器中。

## 8.4 本章小结

在本章中, 你学习了同构应用的优势: 更高的性能、搜索引擎优化、优雅降级(即使在本地禁用了 JavaScript, 应用也能部分正常工作)。现在你了解了如何在服务器端渲染 React 组件, 以及如何“挂载”已经渲染好的 React 组件。



# 测试 React 组件

因为我们的应用变得越来越复杂，而且还在持续不断地增加新功能，因此需要确保新功能在实现时没有为已有功能引入新 bug。自动化测试提供了一个活生生的文档来描述应用的预期行为，能够让我们在开发过程中更有信心，在第一时间就能了解出现的问题。

本章会介绍 Jest(React 推荐的测试框架)和 TestUtils(它提供了一组方法，可在任何一种常见的 JavaScript 测试框架中对 React 组件进行测试)。

## 9.1 Jest

Jest 是 React 推荐的测试框架，它基于流行的 Jasmine 框架并加入了一些很有用的特性：

- 它会在虚拟 DOM 上运行你的测试(所以可以在命令行中运行测试)。
- 内置支持 JSX。

### 1. Jest 测试项目结构

在一个项目中使用 Jest，只需要做两件事：在 `package.json` 中配置一个测试任务；添加一个 `__test__` 文件夹(这是 Jest 测试文件的默认路径)。为演示这个过程，让我们来创建一个新的文件夹并设置这个项目结构。

在新文件夹中，创建一个 `package.json` 项目文件(`npm init -y`)，然后安装 Jest 和 `babel-jest`(`npm install --save-dev jest-cli babel-jest`)。

接下来编辑 `package.json` 文件，使用 Jest 和 `babel-jest` 来设置测试任务。代码清单 9-1 展示了更新后的文件。

代码清单 9-1：使用 `babel-jest` 在 `package.json` 文件中配置测试任务

```
{
  "name": "testsample",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
```

```

    "devDependencies": {
      "babel-jest": "^5.3.0",
      "jest-cli": "^0.6.1"
    },
    "scripts": {
      "test": "jest"
    },
    "jest": {
      "scriptPreprocessor": "<rootDir>/node_modules/babel-jest"
    }
  }
}

```

在代码清单 9-1 中需要注意的是, 使用了<rootDir>作为配置参数, Jest 会在它指定的文件夹中查找测试和模块。在 package.json 中使用该参数时, 这个参数的默认值是 package.json 所在的文件夹。

最后创建一个\_\_tests\_\_文件夹(注意在前后各有两个下划线), 我们就完成了基本的项目结构。

## 2. 开始

在使用 React 之前, 让我们先使用普通的 JavaScript 对象来配置一下 Jest 的测试环境。考虑这样的场景: 在项目根文件夹下有一个 sum.js 文件, 我们要对这个文件进行测试。该文件如代码清单 9-2 所示。

### 代码清单 9-2: sum.js

```

let sum = (value1, value2) => (
  value1 + value2
)
export default sum;

```

在\_\_tests\_\_文件夹中创建一个 sum-test.js 文件, 如代码清单 9-3 所示。

### 代码清单 9-3: sum-test.js

```

jest.autoMockOff();

describe('sum', function() {
  it('adds 1 + 2 to equal 3', function() {
    var sum = require('../sum');
    expect(sum(1, 2)).toBe(3);
  });
});

```

现在可使用在早些时候创建的测试任务来运行测试了(使用 npm test)。输出结果展现在图 9-1 中。

```
$ npm test
> jest-getting-started@1.0.0 test /Users/cassiozen/jest-getting-started
> jest

Using Jest CLI v0.6.1
PASS __tests__/sum_test.js (0.427s)
1 test passed (1 total in 1 test suite, run time 0.872s)
```

图 9-1 测试通过

注意：

在测试文件的第一行，注意到我们禁用了Jest的auto-mocking(使用jest.autoMockOff)。该机制可以允许一个模块独立于其依赖项，这样的话就可以独立地测试每一个代码单元，而不需要依赖相关组件的具体实现。

然而，并非所有代码都可以脱离依赖项进行测试(尤其是那些已经存在的代码文件，它们在编写时可能没有考虑到测试问题)。在这种情况下，更好的策略是禁用 auto-mocking 机制，并针对指定的模块显式地开启它。

9.2 React 测试工具

React 本身就内置一些测试工具，可用于针对组件的测试。这些测试工具作为独立的插件提供，使用 npm 命令 npm install --save-dev react-addons-test-utils 进行安装。

9.2.1 渲染用于测试的组件

React 测试工具中最常用的一个方法是 renderIntoDocument。顾名思义，该方法会将组件渲染到一个脱离文档的独立 DOM 节点中，这样一来可以对生成的 DOM 节点添加一些断言，而不需要把组件插入实际页面中。最基本的使用方式如下：

```
let component = TestUtils.renderIntoDocument(<MyComponent />);
```

然后可使用 findDOMNode()来访问原始的 DOM 元素，并测试其中的值。

使用 renderIntoDocument 和 Jest 的示例

作为示例，下面创建一个新项目，使用 9.1 一节中的“Jest 测试项目结构”部分中的默认结构。在根文件夹下创建一个 CheckboxWithLabel 组件，然后在\_\_tests\_\_文件夹中创建一个 CheckboxWithLabel\_test.js 文件。图 9-2 展示了项目结构。

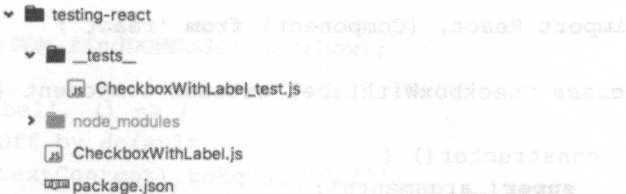


图 9-2 带有测试的 React 项目结构

接下来,更新 `package.json` 文件,向其中加入 `jest` 测试任务配置。它和前面介绍的稍有不同,因为我们要包含一个与 `React` 相关的配置。代码清单 9-4 中展示了更新后的 `package.json`。

代码清单 9-4: `package.json` 文件,使用 `babel-jest` 作为测试配置,进行 `React` 应用测试

```
{
  "name": "testing-react",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-jest": "^5.3.0",
    "jest-cli": "^0.6.1",
    "react": "^0.14.1",
    "react-addons-test-utils": "^0.14.1",
    "react-dom": "^0.14.1"
  },
  "scripts": {
    "test": "jest"
  },
  "jest": {
    "scriptPreprocessor": "<rootDir>/node_modules/babel-jest",
    "unmockedModulePathPatterns": [
      "<rootDir>/node_modules/react",
      "<rootDir>/node_modules/react-dom",
      "<rootDir>/node_modules/react-addons-test-utils",
      "<rootDir>/node_modules/fbjs"
    ]
  }
}
```

`React` 组件非常简单:实现一个简单的复选框,在两个标签之间进行切换,如代码清单 9-5 所示。

代码清单 9-5: `CheckboxWithLabel.js`

```
import React, {Component} from 'react';

class CheckboxWithLabel extends Component {

  constructor() {
    super(...arguments);
    this.state = {isChecked: false};
  }
}
```

```

    this.onChange = this.onChange.bind(this);
  }

  onChange() {
    this.setState({isChecked: !this.state.isChecked});
  }

  render() {
    return (
      <label>
        <input type="checkbox"
          checked={this.state.isChecked}
          onChange={this.onChange} />
        {this.state.isChecked ? this.props.labelOn : this.props.labelOff}
      </label>
    );
  }
}

```

```
export default CheckboxWithLabel;
```

在测试代码中，首先使用 React 的 TestUtils 中的 `renderIntoDocument`，将组件渲染在一个独立的 DOM 节点中。然后，立即使用 `ReactDOM.findDOMNode()` 方法访问组件中原始的 DOM 元素。最后，进行一个断言，预期组件的标签内容会以“off”开头。代码清单 9-6 展示了测试文件。

#### 代码清单 9-6: CheckboxWithLabel\_test.js

```

jest.autoMockOff();

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';

const CheckboxWithLabel = require('../CheckboxWithLabel');

describe('CheckboxWithLabel', () => {
  // Render a checkbox with label in the document
  var checkbox = TestUtils.renderIntoDocument(
    <CheckboxWithLabel labelOn="On" labelOff="Off" />
  );

  var checkboxNode = ReactDOM.findDOMNode(checkbox);

  it('defaults to Off label', () => {
    // Verify that it's Off by default
    expect(checkboxNode.textContent).toEqual('Off');
  });
});

```

通过 `npm test` 来运行这个测试，它应该会通过。注意，我们只能使用 ES6 语法来编写测试，因为我们使用了 `babel-jest`(在 `package.json` 项目文件中作为依赖项加入)，如图 9-3 所示。

```
Using Jest CLI v0.6.1
PASS __tests__/CheckboxWithLabel_test.js (0.865s)
1 test passed (1 total in 1 test suite, run time 1.429s)
```

图 9-3 测试结果

9.2.2 遍历并查找子节点

把组件渲染到 DOM 节点中只是测试 React 组件的第一步，不过在大多数情况下，需要遍历组件的渲染树来查找一个特定的子节点并进行断言。React 的 `TestUtils` 中提供了 6 个函数来实现这一功能，如表 9-1 所示。

表 9-1 在组件渲染树中遍历并查找子节点的工具方法

函数	说明
<code>scryRenderedDOMComponentsWithClass</code>	在组件渲染树中找到所有 class 名匹配 <code>className</code> 的 DOM 组件实例
<code>findRenderedDOMComponentWithClass</code>	与 <code>scryRenderedDOMComponentsWithClass()</code> 类似，不过预期只有一个结果。如果只有一个的话返回它，否则会抛出一个异常
<code>scryRenderedDOMComponentWithTag</code>	在组件渲染树中找到所有标签名称匹配 <code>tagName</code> 的 DOM 组件实例
<code>findRenderedDOMComponentWithTag</code>	与 <code>scryRenderedDOMComponentWithTag()</code> 类似，不过预期只有一个结果。如果只有一个的话就返回它，否则会抛出一个异常
<code>scryRenderedDOMComponentWithType</code>	在组件渲染树中找到所有类型为 <code>componentName</code> 的 DOM 组件实例
<code>findRenderedComponentWithType</code>	与 <code>scryRenderedDOMComponentWithType()</code> 类似，不过预期只有一个结果。如果只有一个的话就返回它，否则会抛出一个异常

让我们在示例项目中添加一个新的测试，使用查找相关的工具方法。使用 `findRenderedDOMComponentWithTag` 方法找到 `input` 元素，验证它不是选中状态。代码清单 9-7 展示了更新后的源代码。

代码清单 9-7：更新后的 `CheckboxWithLabel_test.js`

```
jest.autoMockOff();
import React from 'react';
```



```

import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';

const CheckboxWithLabel = require('../CheckboxWithLabel');

describe('CheckboxWithLabel', () => {

  // Render a checkbox with label in the document
  var checkbox = TestUtils.renderIntoDocument(
    <CheckboxWithLabel labelOn="On" labelOff="Off" />
  );

  var checkboxNode = ReactDOM.findDOMNode(checkbox);

  it('defaults to Off label', () => {
    // Verify that it's Off by default
    expect(checkboxNode.textContent).toEqual('Off');
  });

  it('defaults to unchecked', () => {
    // Verify that the checkbox input field isn't checked by default
    let checkboxElement = TestUtils.findRenderedDOMComponentWithTag(
      checkbox, 'input');
    expect(checkboxElement.checked).toBe(false);
  });
});

```

### 9.2.3 模拟事件

在 React TestUtils 测试工具类中最有用的工具之一是 Simulate 函数，它可以触发用户事件，例如鼠标单击。下面在之前的项目中添加一个新的测试，模拟一次单击来改变 CheckboxWithLabel 中的文本。代码清单 9-8 展示了更新后的测试文件。

代码清单 9-8：更新后的 CheckboxWithLabel\_test.js 源代码

```

jest.autoMockOff();

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';

const CheckboxWithLabel = require('../CheckboxWithLabel');

describe('CheckboxWithLabel', () => {
  // Render a checkbox with label in the document
  var checkbox = TestUtils.renderIntoDocument(
    <CheckboxWithLabel labelOn="On" labelOff="Off" />
  );

```

```

var checkboxNode = ReactDOM.findDOMNode(checkbox);

it('defaults to Off label', () => {
  // Verify that it's Off by default
  expect(checkboxNode.textContent).toEqual('Off');
});

it('defaults to unchecked', () => {
  // Verify that the checkbox input field isn't checked by default
  let checkboxElement = TestUtils.findRenderedDOMComponentWithTag(
    checkbox, 'input');
  expect(checkboxElement.checked).toBe(false);
});

it('changes the label after click', () => {
  // Simulate a click and verify that it is now On
  TestUtils.Simulate.change(
    TestUtils.findRenderedDOMComponentWithTag(checkbox, 'input')
  );
  expect(checkboxNode.textContent).toEqual('On');
});

```

## 9.2.4 浅渲染

浅渲染(Shallow Rendering)是 React 0.13 中引入的一个新特性, 让我们可输出组件的虚拟树, 而不需要生成 DOM 节点。通过这种方法我们可以观察组件是如何构建的, 而不需要真正去渲染它。与使用 `renderIntoDocument` 相比, 它的优势在于去掉了测试环境中对 DOM 的依赖(因此速度也会更快), 此外也能够让我们真正独立地测试一个 React 组件而不必考虑相关的其他组件类。它让我们能测试组件 `render` 方法的返回值, 而不需要对任何子组件进行初始化。

到目前为止, 浅渲染依然只是个试验性质的特性, 不过它已经得到了推动, 并会在未来推荐用于组件测试。

### 1. 基本用法

使用浅渲染非常简单。首先创建一个浅渲染器的实例, 然后用它来渲染组件并获取输出结果。代码清单 9-9 中展示了一个示例实现, 假设你正在测试一个名为 `<MyComponent />` 的组件。

#### 代码清单 9-9: 浅渲染器的基本用法

```

import React from 'react';
import TestUtils from 'react-addons-test-utils';

```

```

const CheckboxWithLabel = require('./MyComponent');

const shallowRenderer = TestUtils.createRenderer();

shallowRenderer.render(<MyComponent className=
  "MyComponent">Hello</MyComponent>);
const component = shallowRenderer.getRenderOutput();

```

它返回了一个表示 React 组件的对象，大致如代码清单 9-10 所示(为简单起见，省略了一些属性)。

代码清单 9-10: 浅渲染器输出的对象

```

{
  "type": "div",
  "props": {
    "className": "MyComponent",
    "children": {
      "type": "h1",
      "props": {
        "children": "Hello "
      }
    }
  }
}

```

现在可在这个组件对象的基础上创建一些断言：

```
expect(component.props.className).toEqual('MyComponent');
```

当你查看浅渲染器返回对象的结构时，可能会注意到它的 `children` 属性。它可能会包含任意文本、DOM 元素或组成正在测试的这个组件的其他 React 组件。

作为演示，把上一个示例中的测试使用浅渲染的方式重写一下。移除原有的测试用例，编写一个新用例，确认复选框默认是未选中状态、标签中的文本是“Off”。代码清单 9-11 展示了更新后的源代码。

代码清单 9-11: 使用 `react-shallow-testutils` 的 `CheckboxWithLabel_test.js` 文件

```

jest.autoMockOff();

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';

const shallowRenderer = TestUtils.createRenderer();
const CheckboxWithLabel = require('../CheckboxWithLabel');

describe('CheckboxWithLabel', () => {

```

```

shallowRenderer.render(<CheckboxWithLabel labelOn="On" labelOff="Off" />);
const checkbox = shallowRenderer.getRenderOutput();

it('defaults to unchecked and Off label', () => {
  // Verify that it's Off by default
  const inputField = checkbox.props.children[0];
  const textNode = checkbox.props.children[1];
  expect(inputField.props.checked).toBe(false);
  expect(textNode).toEqual('Off');
});

});

```

在这个简单的组件中它能很好地运作，不过在遍历那些嵌套层级很深或带有元素数组的组件时，这种方式就显得非常脆弱了。

## 2. React Shallow Test Utils

如上所述，浅渲染目前只处于开发的初步阶段，在 React 0.13 和 0.14 中还缺少一些功能(例如返回已经挂载的组件的实例，以及支持 TestUtils 中的遍历查找函数)。大多数功能会在 React 0.15 中默认提供，不过目前可以安装一个名为 react-shallow-testutils 的 npm 包来实现这些功能。使用 `npm install --save-dev react-shallow-testutils` 进行安装。

react-shallow-testutils 提供的第一个优异的功能就是除了表示组件的对象外，还允许访问已挂载的组件。

下面用一个示例来重写上一个测试。这里不再手动引用元素数组，而在测试中创建 render 方法的预期输出，并与实际组件进行比较。代码清单 9-12 展示了更新后的测试。

### 代码清单 9-12: 完成同一测试的另一种方法

```

jest.autoMockOff();

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
import ShallowTestUtils from 'react-shallow-testutils';

const shallowRenderer = TestUtils.createRenderer();
const CheckboxWithLabel = require('../CheckboxWithLabel');

describe('CheckboxWithLabel', () => {

  // Render a checkbox with label in the document
  shallowRenderer.render(<CheckboxWithLabel labelOn="On" labelOff="Off" />);

  const checkbox = shallowRenderer.getRenderOutput();
  const component = ShallowTestUtils.getMountedInstance(shallowRenderer);

```

```

it('defaults to unchecked and Off label', () => {
  const expectedChildren = [
    <input type="checkbox" checked={false} onChange=
      {component.onChange} />,
    "Off"
  ];
  expect(checkbox.props.children).toEqual(expectedChildren);
});
});

```

在这个示例中你可能注意到了 `onChange` 方法中的引用：`onChange={component.onChange}`。

这里使用 `react-shallow-testutils` 提供的已挂载的组件实例，然后和 `React` 组件使用同一个函数得到的结果进行对比测试。

如果需要在组件状态变更结束后调用 `mountedInstance` 方法，请确保再次调用 `shallowRenderer.getRenderOutput` 来更新渲染结果。下面在示例项目中演示这个过程，调用组件的 `onChange` 方法。代码清单 9-13 展示了更新后的文件。

#### 代码清单 9-13：在已挂载的组件上调用方法

```

jest.autoMockOff();

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
import ShallowTestUtils from 'react-shallow-testutils';

const shallowRenderer = TestUtils.createRenderer();
const CheckboxWithLabel = require('../CheckboxWithLabel');

describe('CheckboxWithLabel', () => {

  // Render a checkbox with label in the document
  shallowRenderer.render(<CheckboxWithLabel labelOn="On" labelOff="Off" />);

  let checkbox = shallowRenderer.getRenderOutput();
  const component = ShallowTestUtils.getMountedInstance(
    shallowRenderer);

  it('defaults to unchecked and Off label', () => {
    const expectedChildren = [
      <input type="checkbox" checked={false} onChange=
        {component.onChange} />,
      "Off"
    ];
    expect(checkbox.props.children).toEqual(expectedChildren);
  });
});

```

```
it('changes the label after click', () => {  
  component.onChange();  
  checkbox = shallowRenderer.getRenderOutput();  
  expect(checkbox.props.children[1]).toEqual('On');  
});  
});
```

使用 npm test 进行测试, 结果如图 9-4 所示。

```
$ npm test  
  
> testing-react@1.0.0 test /Users/cassiozen/testing-react  
> jest  
  
Using Jest CLI v0.6.1  
PASS __tests__/CheckboxWithLabel_test.js (0.76s)  
2 tests passed (2 total in 1 test suite, run time 1.189s)
```

图 9-4 两个测试均已通过

## 9.3 本章小结

在本章你了解到如何使用 React 的测试工具进行组件测试。可在一个脱离页面的 DOM 节点中生成组件的 DOM 树(使用 `renderIntoDocument` 方法), 也可以使用浅渲染方式在不进行任何渲染的情况下输出组件的虚拟树。组件呈现后, 可使用任何一个测试框架来设置断言(用于组件的属性、节点等)。你同样学习了 Jest 这个测试框架, 它由 Facebook 提供, 并推荐用于 React 项目。



# JavaScript 2015

JavaScript 2015(也被称为 ECMAScript 6, 简称 ES6)是 JavaScript 语言最新的版本。为了编写复杂的应用, 它加入了很多重要的新语法, 包括类、模块、新的变量声明关键字和 Promise。它还包含了一些帮助方法和函数级别的语法糖, 用来让你的代码更具表达性, 例如, 箭头函数、模板字符串和解构赋值。

## 类

JavaScript 中的类是在 ECMAScript 6 中引入的, 它实际上是一个语法糖, 背后是 JavaScript 中已有的基于原型链的继承体系。JavaScript 中类的语法并非是一套新的面向对象的模型, JavaScript 中的类语法在创建对象、处理继承的时候要更加简洁和清晰。

类支持原型链继承、父级调用、实例方法、静态方法以及构造函数。

```
class Point {
  constructor(x, y) {
    this.x= x;
    this.y= y;
  }
  toString() {
    return `${this.x}, ${this.y}`;
  }
}

class Pixel extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color= color;
  }
  toString() {
    return super.toString() + ' in ' + this.color;
  }
}

const p = new Pixel(25,8,'green');
p.toString();// (25, 8) in green
```

到目前为止，ES6 中的类在本质上并没有什么新功能。它只是提供了更方便的语法来提供原有的构造函数的功能。通过键入以下代码可以证实这一点：

```
typeof Point // 'function'
```

## 模块

对于任何一种编程语言来说，模块都是最重要的特性之一。模块在 ES6 中属于一等公民。ES6 的模块是一个包含了 JS 代码的文件。在 ES6 中，并没有特殊的 `module` 关键字，模块通常看上去就像是一段普通脚本，只是其中带有 `export` 关键字。

### export

首先让我们了解一下 `export`。在模块中声明的所有内容对该模块来说，默认情况下都是本地的(local)。如果你需要在模块内声明一些公有的内容，从而让其他模块可以使用它们的话，必须“导出(export)”该功能。

```
function generateRandom() {
  return Math.random();
}

function sum(a, b) {
  return a + b;
}

export { generateRandom, sum }
```

你可以 `export` 任何处于顶级作用域的函数、类、变量或常量。

### import

`import` 语句用来导入函数、对象，或是来自其他模块或脚本导出的基元。

```
import { generateRandom, sum } from 'utility';

console.log(generateRandom()); // logs a random number
console.log(sum(1, 2)); // 3
```

### 默认导出

默认导出可以用于从模块中导出单个值：

```
var utils = {
  generateRandom: function() {
    return Math.random();
  },
  sum: function(a, b) {
    return a + b;
  }
}
```

```

    }
};

```

```
export default utils;
```

导入的时候简单地使用默认导出的名称即可：

```

import utils from 'utility';
console.log(utils.generateRandom()); // logs a random number
console.log(utils.sum(1, 2)); // 3

```

## let 和 const

使用“var”声明的变量拥有一个潜在的风险：它只在函数的作用域范围内是本地变量，为说明这一点，请看下面的示例：

```

function foo() {
    var myVariable = "10";
}

foo()
console.log(myVariable)
// 和预期的一样，这句话会抛出一个“未定义”异常，这个变量被限制在函数的作用域内

```

然而，当它在一个语句块中声明时，这个变量是位于全局作用域的：

```

if (true) {
    var myVariable = "10";
}

console.log(myVariable)
// 会输出“10”，因为 myVariable 变量并非在“if”语句块的作用域内，而是全局的

```

### let

在声明变量时，ES6 中引入了两个新的关键字：let 和 const。这两个都是用来创建块级作用域的内容。let 相当于新的 var。

```

if (true) {
    let myVariable = "10";
}

console.log(myVariable)
// 会抛出一个“未定义”异常，该变量是位于“if”的块作用域的

```

通常来说，在任何需要使用 var 的方改为使用 let 都会更安全一些。

### const

const 声明是个一次性的赋值：它创建了针对一个值的只读引用。它并不意味着其中

包含的值是不可变的，只不过这个标识不能够被再次赋值而已。

```
const MY_FAV = 7; // 定义 MY_FAV 作为常量，并赋值为 7
MY_FAV = 20; // 会抛出一个异常：“试图为常量赋值”
```

## Promise

Promise 是一个异步编程的库，在未来，Promise 可能会成为语法中的一等公民。Promise 库在大量的 JavaScript 库中都有着广泛的应用。

```
function timeout(duration = 0) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, duration);
  })
}

var p = timeout(1000).then(() => {
  return timeout(2000);
}).then(() => {
  throw new Error("hmm");
}).catch(err => {
  return Promise.all([timeout(100), timeout(200)]);
})
```

## 箭头函数

箭头函数是使用 `=>` 语法来定义函数的缩写形式，它主要为了实现两个目的：更简洁的语法；和父作用域共享 `this` 关键字。

### 简洁的语法

经典的 JavaScript 函数语法没有任何灵活性可言，不管是只包含一条语句的函数，还是不幸长达数页的函数都是一样的定义。每次你需要定义函数的时候都需要键入烦人的 `function() {}`。箭头函数的语法最主要的好处之一就是提供了简洁的函数定义方式：

```
// 简单的示例
setInterval(() => console.log("Time is passing"), 1000);

// 单行 (隐式返回)
let square = (num) => num * num;
console.log(square(5)) // 返回 25

// Multiple lines
nums.forEach(v => {
  if (v % 5 === 0)
```

```
fives.push(v);
});

// 多行和隐式返回
let actors = ['Adam West', 'Michael Keaton', 'Val Kilmer', 'George Clooney', 'Christian Bale', 'Ben Affleck']
actors.map((actor)=>(
  actor +' was Batman!\n'
));
```

## 词法绑定

在 JavaScript 中，每个函数都定义了自己的 `this` 上下文，它很简洁，也很烦人。箭头函数的词法绑定会把 `this` 绑定到它的声明所在父级作用域的上下文中：

```
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name+" knows "+ f));
  }
}
```

## 模板字符串

JavaScript 中的字符串因为历史原因有着种种局限，缺少了很多像是 Python 或 Ruby 语言中的一些特性。

ES6 中的模板字符串彻底改变了这一现状，它提供了用于构造字符串的语法糖。

## 语法

模板字符串使用反引号(`)而不是通常在字符串中使用的单引号或双引号。模板字符串写起来是这样的：

```
var greeting = `Yo World!`;
```

到目前为止，模板字符串还没有带给我们任何普通字符串之外的功能，让我们来看看它能做些什么。

## 字符串置换

模板字符串首先最大的优势就是用于字符串置换，它可以让我们在一个模板声明内部使用任何 JavaScript 表达式(包括变量)，其结果会作为该字符串的一部分进行替换输出。

模板字符串可以使用`\${}`语法作为字符串置换的占位符，例如：

```
// 简单的字符串置换
var name = "Brendan";
console.log(`Yo, ${name}!`);

// => "Yo, Brendan!"
```

## 多行字符串

在 JavaScript 中使用多行字符串有时需要一些旁门左道的变通方式。而模板字符串在处理多行字符串的时候就非常简单，直接在里面包含换行即可，例如：

```
console.log(`string text line 1
string text line 2`);
```

## 解构赋值

解构赋值语法是一个 JavaScript 表达式，用于从数组或对象中提取出不同的变量。

```
// 数组匹配
var a, b, rest;
[a, b] = [1, 2]
console.log(a) // 1
console.log(b) // 2
```

```
// 对象匹配
var robotA = {name: "Bender"};
var robotB = {name: "Flexo"};

var {name: nameA} = robotA;
var {name: nameB} = robotB;
```

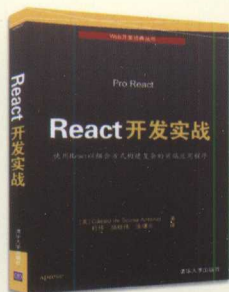
```
console.log(nameA);
// "Bender"
console.log(nameB);
// "Flexo"
```

```
// 也可以在参数中使用
functiong({name: x}) {
  console.log(x);
}

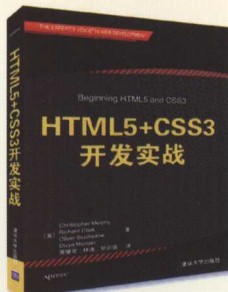
g({name: 5})
```



## Web开发经典丛书

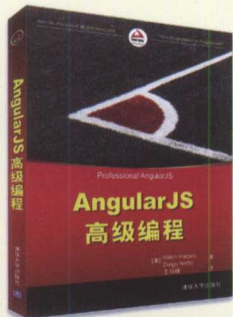


本书介绍如何成功构建日益复杂的前端应用程序与接口，深入分析React库，并详述React生态系统中的其他工具与库，从而指导你创建完整的复杂应用程序。你将全面学习React的用法以及React生态系统中的其他工具和库(如React Router和Flux架构)，并了解采用组合模式创建接口的最佳实践。如果你拥有一些使用jQuery或其他JavaScript框架来创建前端应用程序的经验，但想解决复杂前端应用程序构建过程中日益增多的常见问题，那么本书就是为你准备的。



本书循序渐进地介绍了HTML5的特性与元素，将帮助你充分利用HTML5所提供的各种更为简洁、清晰且高效的代码，并向你展示如何使用CSS3的结构整体性和样式灵活性——这意味着你的网站项目能有更好的页面外观和更漂亮的内容。

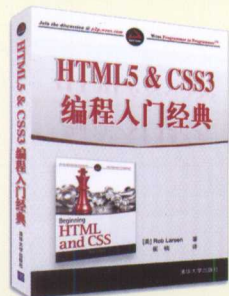
使用本书提供的实用而循序渐进的方法，可以让你的Web开发跻身于先进行列。本书提供了你所需要的知识与技能，以接纳新的Web标准以及HTML5与CSS3的最新特性。



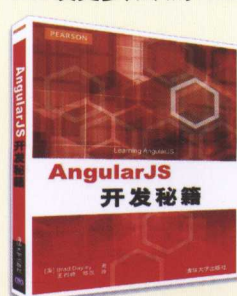
本书内容广泛，涵盖构建首个AngularJS应用乃至构建系统和自动执行集成测试等高级主题。如果你已经熟悉基本编程知识，并希望编写在浏览器UI中显示复杂数据的Web应用，或快捷高效地模拟本地应用的移动Web应用，那么本书将是值得你反复研读的必备指南。



从5年前推出本书第1版以来，jQuery经历了大幅修改和增强。本书涵盖所有新内容和增强内容，透彻讲述新的HTML5元素和功能、改进的事件处理方法以及升级后的jQuery UI等。每章都帮助读者学习通过jQuery易用的卓越功能来开发动态Web页面和Web应用程序。本书是有志于学习JavaScript、CSS及更多知识的Web开发人员的必备书籍。



HTML和CSS是构建网页所需要了解的两核心编程语言，本书详细介绍了这两种语言。本书提供了对于最佳实践及技术的手把手指导。作为一本实用参考，本书深入阐述了为当今多设备多平台环境开发独特的现代网站所需的HTML及CSS最新版本。本书涵盖了丰富的内容：从为网页组织文档结构以及微调文本，到链接到其他网页或电子邮件地址，以及使用图片、音频、视频和表格。



AngularJS是Web开发领域最激动人心的创新技术之一，它为整个开发过程提供结构，旨在简化Web应用的开发和测试。本书展示如何创建功能强大的交互性Web应用，这些应用具有结构良好、便于维护、可重用的基本代码。你还将学习如何使用AngularJS的创新MVC模式来开发结构和设计俱佳的网页和Web应用。



《React开发实战》介绍如何成功构建日益复杂的前端应用程序与接口，深入分析React库，并详述React生态系统中的其他工具与库，从而指导你创建完整的复杂应用程序。

你将全面学习React的用法以及React生态系统中的其他工具和库(如React Router和Flux架构)，并了解采用组合方式创建接口的最佳实践。本书简明扼要地讲解每个主题，并呈现助你高效完成工作的细节。书中严谨深刻地讲述React中最重要的功能，每章还详细列出常见的开发问题，并解释如何避免它们。

如果你拥有使用jQuery或其他JavaScript框架创建前端应用程序的经验，但想解决复杂前端应用程序构建过程中日益增多的常见问题，那么本书就是为你准备的。开始像专家那样去使用React吧，今天就把这本书收入囊中！

## 主要内容

- ◆ 如何创建可组合的用户界面
- ◆ 理解React的虚拟DOM架构以及如何利用该架构开发应用程序
- ◆ 了解各项功能的原理及重要性
- ◆ 深入学习React以及React生态系统中重要的第三方库
- ◆ 学习如何创建通用/同构应用程序从而改进用户体验和SEO
- ◆ 深刻理解复杂应用程序中的数据流策略
- ◆ 学习如何测试、完善和部署React项目

清华大学出版社数字出版网站

WQBook   
www.wqbook.com

Apress®  
www.apress.com



源代码下载

ISBN 978-7-302-46197-5



9 787302 461975 >

定价：58.00元